pyH2A

Release 0.0a8

Jacob Schneidewind

Nov 07, 2022

CONTENTS

CHAPTER

ONE

GETTING STARTED

Table of Contents

1.1 Installation

pyH2A can be installed using `pip`:

```
pip install pyH2A
```

1.2 Choose configuration

First, the configuration of the hydrogen production technology should be specified. This is done by selecting the appropiate plugins, which together form the desired production pathway. For example, in case of photovoltaic + electrolysis (PV+E), the *Hourly_Irradiation_Plugin* and *Photovoltaic_Plugin* may be used: *Hourly_Irradiation_Plugin* models the irradiation in specified location, while *Photovoltaic_Plugin* models electricity production using PV based on the hourly irradiation data and subsequent production of hydrogen from electrolysis. Changing the plugins changes the technology configuration and new configurations (e.g. including battery storage) can be modelled by creating new plugins (see *Plugin Guide* for information on how to create new plugins).

The chosen plugins are specified in the *Workflow* table of the input file:

```
# Workflow

Name | Type | Position
--- | --- | ---
Hourly_Irradiation_Plugin | plugin | 0
Photovoltaic_Plugin | plugin | 0
```

`Name` is the name of the used module, `Type` is the type of module (in both cases `plugin`) and `Posiition` refers to the the position in the workflow when this module is executed (in this case both positions are 0, meaning that these plugins are executed at the beginning of the workflow with `Hourly_Irradiation_Plugin` being executed before `Photovoltaic` plugin). See *Default Settings* for the default positions of the different elements of the workflow.

At this point one may also specify the analysis modules which are to be used. These are included by putting a header with the name of the analysis module into the input file.

```
# Monte_Carlo_Analysis
```

This header request the *Monte_Carlo_Analysis* module.

1.3 Generate input file template

The input file containing the `Workflow` tabe and possible analysis headings is the starting point to generate the full input file template.

At this point the input file may look like this:

```
# Workflow

Name | Type | Position
--- | --- | ---
Hourly_Irradiation_Plugin | plugin | 0
Photovoltaic_Plugin | plugin | 0

# Monte_Carlo_Analysis
```

In the current directory, the `generate` function from the pyH2A command line interface may be used to generate the full input file template:

```
pyH2A generate -i input.md -o input_full.md --origin --comments
```

The `--origin` flag includes information in the template on which plugin/module has requested a given input. The `--comments` flag includes additional information on the requested input (from the documentation). The flags can be omitted to obtain a cleaner input file template.

The thus generated file `input_full.md` can be used to enter the model information.

1.4 Enter model information

The input file template specifies which model information has to be entered for the selected technology configuration. For example, `Hourly_Irradiation_Plugin` requests a file containg hourly irradiation data:

```
# Hourly Irradiation

Parameter | Value | Comment Value
--- | --- | ---
File | str | Path to a `.csv` file containing hourly irradiance data as provided by␣
↪https://re.jrc.ec.europa.eu/pvg_tools/en/#TMY, ``process_table()`` is used.
```

`str` indicates that a string which a path to the file is requested (regular Python types are used for input prompts, such as `str`, `int`, `float`, `ndarray` etc.).

Other tables allow for flexible processing of input information, which is indicated by the placeholder `[...]`. For example, the default `Capital_Cost_Plugin` creates this input prompt:

```
# [...] Direct Capital Cost [...]

Parameter | Value | Comment Value
--- | --- | ---
[...] | float | ``sum_all_tables()`` is used.
```

The leading and ending `[...]` indicates a table group, meaning that all tables containing the center string in their heading will be processed together (in case of `Direct Capital Cost` this can for example be used to break up the information on direct capital costs into seperate tables for easier readability and subsequent cost breakdown analysis).

The `[...]` in the `Parameter` column indidcates that any parameter name can be chosen here and any number of parameters can be entered into the table. *`sum_all_tables()`* means that all the information will ultimately be summed up to compute the total capital cost.

Instead of entering actual values, it is also possible to enter references to other parts of the input file, using the `table > row > column` synthax. This kind of reference can either be entered directly into the prompted input field (for example entering it in the `Value` column of `Direct Capital Cost` table), or `Path` column can be added. For example:

```
# Electrolyzer

Name | Value
--- | ---
Nominal Power (kW) | 5,500.0
...

# Photovoltaic

Name | Value | Path
--- | --- | ---
Nominal Power (kW) | 1.5 | Electrolyzer > Nominal Power (kW) > Value
...
```

In this case, the `Path` column of `Photovoltaic > Nominal Power (kW) > Value` references `Electrolyzer > Nominal Power (kW) > Value`. Because the reference is in the `Path` column, the referenced value is multiplied by the value in `Photovoltaic > Nominal Power (kW) > Value`. In this case, use of referencing ensures that the photovoltaic nominal power is a factor of 1.5 higher than the electrolyzer nominal power (and it is automatically updated when the electrolyzer nominal power is changed).

1.5 Run pyH2A

Once all the model information has been entered, pyH2A can be run to perform the actual techno-economic analysis. This can be done using the command line interface:

```
pyH2A run -i input_full.md -o .
```

`-i` specifies the path of the input file (in this example the input file is in the current directory) and `-o` specifies the output directory (`.` means the current directory is selected for the output).

Upon completion, pyH2A prints the levelized cost of hydrogen, for example:

```
Levelized cost of hydrogen (base case): 3.5777931317137512 $/kg
```

1.6 Generate plots, save results, access information

The power of pyH2A lies in the ability to interface the core techno-economic analysis with different analysis modules to perform in-depth analysis of the results. For example, when the `Monte_Carlo_Analysis` module is requested in the input file, Monte Carlo analysis is performed in which selected input parameters are randomly varied to analyze the future hydrogen cost trajectory. Typically, analysis modules contain methods to generate plots of the analysis results. These are requested by adding a `Methods` table to the input file. For example:

```
# Methods - Monte_Carlo_Analysis

Name | Method Name | Arguments
--- | --- | ---
distance_cost_relationship | plot_distance_cost_relationship | Arguments - MC Analysis -␣
↪distance_cost
```

Including this table in the input file requests that the *plot_distance_cost_relationship()* method is executed. Arguments can be passed to the method in the `Arguments` column. In this case, a simple string is included `Arguments - MC Analysis - distance_cost`. This directs pyH2A to another table in the input file which contains the method arguments:

```
# Arguments - MC Analysis - distance_cost

Name | Value
--- | ---
show | True
save | False
legend_loc | upper right
log_scale | False
plot_kwargs | {'dpi': 300, 'left': 0.09, 'right': 0.5, 'bottom': 0.15, 'top': 0.95, 'fig_
↪width': 9, 'fig_height': 3.5}
table_kwargs | {'ypos': 0.5, 'xpos': 1.05, 'height': 0.5}
image_kwargs | {'path': 'pyH2A.Other~PV_E_Clipart.png', 'x': 1.6, 'zoom': 0.095, 'y': 0.
↪2}
```

This synthax is useful when a number of arguments are provided. Alternatively, a dictionary which arguments can be directly included in the `Arguments` column:

```
# Methods - Monte_Carlo_Analysis

Name | Method Name | Arguments
--- | --- | ---
distance_cost_relationship | plot_distance_cost_relationship | {'show': True, 'save':␣
↪True}
```

By setting `save` to `True`, the plot is saved to the output directory. In this case, the following plot is generated:

To access detailed information, which is generated during runtime, pyH2A can also be run from a Python script, which allows for full access to the information. For example:

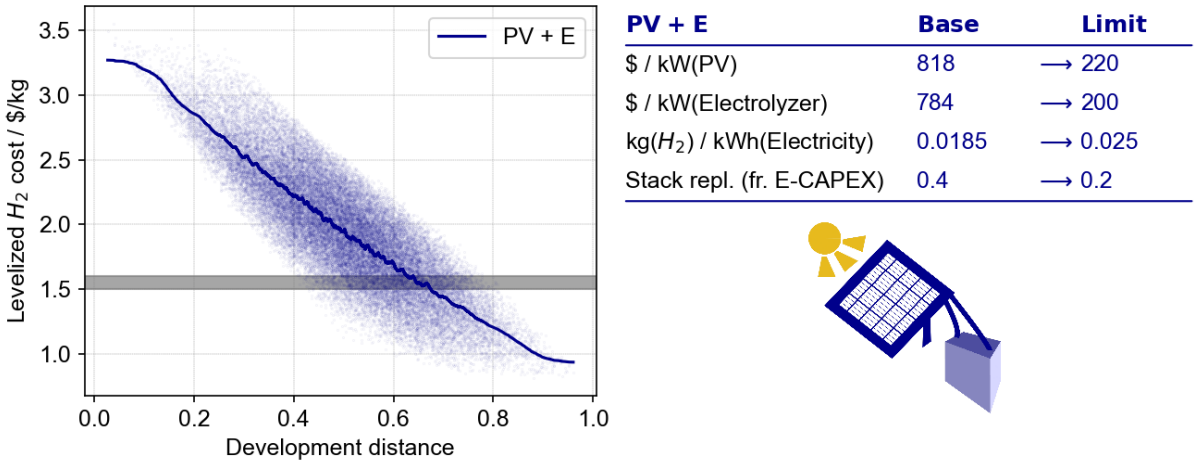

Fig. 1: Example output plot from Monte Carlo analysis.

```
from pyH2A.run_pyH2A import pyH2A

result = pyH2A('input_full.md', '.')
```

`result` is a pyH2A class object. Its attributes contain all the information from the pyH2A run. For example, `result.inp` is a dictionary with all processed input information, `result.base_case` contains the information from the discounted cashflow calculation for the specified input information (base case), including all information generated by plugins (accessible via `result.base_case.plugs`, which is dictionary with all plugin class instances). Furthermore, `result.meta_modules` is a dictionary which contains all of the analysis module class instances, which were generated during the pyH2A run. With this methodology, pyH2A calculations and results can be integrated into other scripts/programs.

CHAPTER

TWO

PYH2A

2.1 cli_pyH2A

2.2 run_pyHA

Functions:

command_line_pyH2A(input_file, output_dir)	Wrapper function to run pyH2A using click.
run_pyH2A()	Wrapper function to run pyH2A from the command line.

Classes:

pyH2A(input_file, output_directory[, print_info])	pyH2A class that performs discounted cash flow analysis and executes analysis modules.

pyH2A.run_pyH2A.**command_line_pyH2A**(*input_file*, *output_dir*)

Wrapper function to run pyH2A using click.

class pyH2A.run_pyH2A.**pyH2A**(*input_file*, *output_directory*, *print_info=False*)

pyH2A class that performs discounted cash flow analysis and executes analysis modules.

Parameters

input_file
[str] Path to input file.

output_directory
[str] Path to output file.

print_info
[bool, optional] Flag to control if detailed information during run of pyH2A is printed.

Returns

pyH2A
[object] pyH2A class object.

Attributes

inp
[dict] Dictionary containing input information and all information from discounted cash flow analysis.

base_case
[Discounted_Cash_Flow object] Discounted_Cash_Flow object for base case defined in input file with corresponding attributes.

meta_modules
[dict] Dictionary containing class instances of executes analysis modules.

Methods:

check_for_plotting_method(method_name)	Returns true if a plotting indicator substring is in *method_name*.
execute_meta_module(module_name, meta_dict)	Requested module class is executed.
execute_module_methods(module, key, ...)	Requested methods of module class are executed.
get_arguments(table)	Arguments are read from the table in self.inp referenced in table['Arguments'] or directly read from table['Arguments']
meta_workflow(meta_dict)	Meta modules (analysis modules) are identified and executed

check_for_plotting_method(*method_name*)

Returns true if a plotting indicator substring is in *method_name*.

execute_meta_module(*module_name*, *meta_dict*)

Requested module class is executed.

execute_module_methods(*module*, *key*, *module_name*, *meta_dict*)

Requested methods of module class are executed.

get_arguments(*table*)

Arguments are read from the table in self.inp referenced in table['Arguments'] or directly read from table['Arguments']

meta_workflow(*meta_dict*)

Meta modules (analysis modules) are identified and executed

Notes

Naming convention for analysis module: in self.inp, the table title has to contain *Analysis* and the last part of the string (seperated by spaces) has to be the module name

pyH2A.run_pyH2A.**run_pyH2A**()

Wrapper function to run pyH2A from the command line.

2.3 Discounted_Cash_Flow

Classes:

Discounted_Cash_Flow(input_file[, ...])	Class to perform discounted cash flow analysis.

Functions:

MACRS_depreciation(plant_years, ...)	Calculation of MACRS depreciations.
discounted_cash_flow_function(inp, values, ...)	Wrapper function for `Discounted_Cash_Flow`, substituting provided values at specified parameter positions and returning desired attribute of `Discounted_Cash_Flow` object.
get_idx(diagonal_number, axis0, axis1)	Calculation of index for MACRS calculation.
numpy_npv(rate, values)	Calculation of net present value.

class pyH2A.Discounted_Cash_Flow.**Discounted_Cash_Flow**(*input_file*, *print_info=True*, *check_processing=True*)

Class to perform discounted cash flow analysis.

Parameters

input_file
[str or dict] Path to input file or dictionary containing input file data.

print_info
[bool] Boolean flag to control if detailed info on action of plugins is printed.

check_processing
[bool] Boolean flag to control if *check_processing* is run at the end of discounted cash flow analysis, which checks if all tables in input file have been processed during run.

Returns

Discounted_Cash_Flow
[object] Discounted cash flow analysis object.

Notes

Numerical inputs

Numbers use decimal points. Commas can be used as thousands seperator, they are removed from numbers during processing. the "%" can be used after a number, indicating that it will be divided by 100 before being used.

Special symbols

Tables in the input file are formatted using GitHub flavoured Markdown. This means that "#" at the beginning of a line indicates a header. "|" is used to seperate columns. "—" is used on its own line to seperate table headers from table entries.

Paths to locations in the input file/in self.inp are specified using ">". Paths are always composed of three levels: top key > middle key > bottom key.

File name paths are specified using "/".

In cases where multiple numbers are used in one field (e.g. during sensitivity analysis), these numbers are seperated using ";".

Order in the input file

Order matters in the following cases:

1. For a group of `sum_all_tables()` processed tables (sharing the specified part of their key, e.g. "Direct Capital Cost"), they are processed in their provided order.

2. Within a table, the first column will be used to generate the "middle key" of self.inp. The order of the other columns is not important.

Input

Processed input cells can contain either a number or path(s) (if multiple paths are used, they have to be seperated by ";") to other cells. The use of process_input() (and hence, process_table(), sum_table() and sum_all_tables()) also allows for the value of an input cell to be multiplied by another cell by including path(s) in an additiona column (column name typically "Path").

Workflow

Workflow specifies which functions and plugins are used and in which order they are executed. The listed five functions have to be executed in the specified order for pyH2A to work. Plugins can be inserted at appropiate positions (Type: "plugin"). Plugins have to be located in the ./Plugins/ directory. Execcution order is determined by the "Position" input. If the specified position is equal to an already exisiting one, the function/plugin will be executed after the already specified one. If multiple plugins/function are specified with the same position, they will be executed in the order in which they are listed in the input file.

Plugins

Plugins needs to be composed of a class with the same name as the plugin file name. This class uses two inputs, a discounted cash flow object (usually indicated by "self") and "print_info", which controls the printing of run time statements. Within the __init__ function of the class the actions of the plugins are specified, typically call(s) of the "insert()" function to modify the discounted cash flow object's "inp" dictionary (self.inp).

Attributes

h2_cost
: [float] Calculate levelized H2 cost.

contributions
: [dict] Cost contributions to H2 price.

plugs
: [dict] Dictionary containing plugin class objects used during analysis.

Methods:

cash_flow()	Calculate cash flow.
check_processing()	Check whether all tables in input file were used.
cost_contribution()	Compile contributions to H2 cost.
debt_financing()	Calculate constant debt financing.
depreciation_charge()	Calculate depreciation charge.
execute_function(function_name, npv_dict)	Execute class function named *function_name* and store output in *npv_dict*
expenses_per_kg_H2(value)	Calculate expenses per kg H2.
fixed_operating_costs()	Calculate fixed operating costs.
h2_cost()	Calculate levelized H2 cost.
h2_revenue()	Calculate H2 sales revenue.
h2_sales()	Calculate H2 sales.
income()	Calculate total income.
inflation()	Calculate inflation correction and inflators for specific commodities.
initial_equity_depreciable_capital()	Calculate initial equity depreciable capital.
non_depreciable_capital_costs()	Calculate non-depreciable capital costs.
post_workflow()	Functions executed after workflow.
pre_workflow()	Functions executed before workflow.
production_scaling()	Get plant outpuer per year at gate.
replacement_costs()	Calculate replacement costs.
salvage_decommissioning()	Calculate salvage and decomissioning costs.
time()	Creating time scale information for discounted cash flow analysis.
variable_operating_costs()	Calculate variable operating costs.
workflow(inp, npv_dict, plugs_dict)	Executing plugins and functions for discounted cash flow.
working_capital_reserve_calc()	Calculate working capital reserve.

cash_flow()

Calculate cash flow.

check_processing()

Check whether all tables in input file were used.

Notes

'Workflow' and 'Display Parameters' tables are exempted. Furthermore, all tables that have the term 'Analysis' in their name are also exempted.

cost_contribution()

Compile contributions to H2 cost.

debt_financing()

Calculate constant debt financing.

depreciation_charge()

Calculate depreciation charge.

execute_function(*function_name*, *npv_dict*)

Execute class function named *function_name* and store output in *npv_dict*

`expenses_per_kg_H2`(*value*)

Calculate expenses per kg H2.

`fixed_operating_costs`()

Calculate fixed operating costs.

Parameters

Financial Input Values > startup time > Value
[int] Startup time in years.

Financial Input Values > startup cost fixed > Value
[float] Percentage of fixed operating costs during start-up.

Fixed Operating Costs > Total > Value
[float] Total fixed operating costs.

`h2_cost`()

Calculate levelized H2 cost.

`h2_revenue`()

Calculate H2 sales revenue.

`h2_sales`()

Calculate H2 sales.

`income`()

Calculate total income.

`inflation`()

Calculate inflation correction and inflators for specific commodities.

`initial_equity_depreciable_capital`()

Calculate initial equity depreciable capital.

Parameters

Financial Input Values > equity > Value
[float] Percentage of equity financing.

Financial Input Values > irr > Value
[float] After tax real internal rate of return.

Depreciable Capital Costs > Inflated > Value
[float] Inflated depreciable capital costs.

`non_depreciable_capital_costs`()

Calculate non-depreciable capital costs.

Parameters

Non-Depreciable Capital Costs > Inflated > Value
[float] Inflated non-depreciable capital costs.

`post_workflow`()

Functions executed after workflow.

Parameters

Financial Input Values > depreciation length > Value
[int] Depreciation length in years.

Financial Input Values > depreciation type > Value
[str] Type of depreciation, currently only MACRS is implemented.

Financial Input Values > interest > Value
[float] Interest rate on debt.

Financial Input Values > debt > Value
[str] Debt period, currently only constant debt is implemented.

Financial Input Values > startup revenues > Value
[float] Percentage of revenues during start-up.

Financial Input Values > decommissioning > Value
[float] Decomissioning cost in percentage of depreciable capital investment.

Financial Input Values > salvage > Value
[float] Salvage value in percentage of total capital investment.

Financial Input Values > state tax > Value
[float] State tax.

Financial Input Values > federal tax > Value
[float] Federal tax.

Financial Input Values > working capital > Value
[float] Working capital as percentage of yearly change in operating costs.

`pre_workflow()`

Functions executed before workflow.

Parameters

Workflow > initial_equity_depreciable_capital > Type
[function] Initial equity depreciable capital function.

Workflow > initial_equity_depreciable_capital > Position
[int] Position of initial equity depreciable capital function.

Workflow > non_depreciable_capital_costs > Type
[function] Non-depreciable capital costs function.

Workflow > non_depreciable_capital_costs > Position
[int] Position of non-depreciable capital costs function.

Workflow > replacement_costs > Type
[function] Replacement costs function.

Workflow > replacement_costs > Position
[int] Position of replacement costs function.

Workflow > fixed_operating_costs > Type
[function] Fixed operating costs function.

Workflow > fixed_operating_costs > Position
[int] Position of fixed operating costs function.

Workflow > variable_operating_costs > Type
[function] Variable operating costs function.

Workflow > variable_operating_costs > Position
[int] Position of variable operating costs function.

Workflow > [...] > Type
[plugin, optional] Plugin to be executed.

Workflow > [...] > Position
[int, optional] Position of plugin to be executed.

Financial Input Values > ref year > Value
[int] Financial reference year.

Financial Input Values > startup year > Value
[int] Startup year for plant.

Financial Input Values > basis year > Value
[int] Financial basis year.

Financial Input Values > current year capital costs > Value
[int] Current year for capital costs.

Financial Input Values > plant life > Value
[int] Plant life in years.

Financial Input Values > inflation > Value
[float] Inflation rate.

Construction > [...] > Value
[float] Percentage of capital spent in given year of construction. Number of entries determines construction period in year (each entry corresponds to one year). Have to be in order (first entry corresponds to first construction year etc.). Values of all entries have to sum to 100%.

Returns

Financial Input Values > construction time > Value
[int] Construction time in years.

`production_scaling()`

Get plant outpuer per year at gate.

Parameters

Technical Operating Parameters and Specifications > Output per Year at Gate > Value
[float] Output per year at gate in kg.

`replacement_costs()`

Calculate replacement costs.

Parameters

Replacement > Total > Value
[ndarray] Total replacement costs.

`salvage_decommissioning()`

Calculate salvage and decomissioning costs.

`time()`

Creating time scale information for discounted cash flow analysis.

`variable_operating_costs()`

Calculate variable operating costs.

Parameters

Financial Input Values > startup cost variable > Value
[float] Percentage of variable operating costs during start-up.

Variable Operating Costs > Total > Value
[ndarray] Total variable operating costs.

workflow(*inp*, *npv_dict*, *plugs_dict*)

Executing plugins and functions for discounted cash flow.

working_capital_reserve_calc()

Calculate working capital reserve.

pyH2A.Discounted_Cash_Flow.**MACRS_depreciation**(*plant_years*, *depreciation_length*, *annual_depreciable_capital*)

Calculation of MACRS depreciations.

Parameters

plant_years
[ndarray] Array of plant years.

depreciation_length
[int] Depreciation length.

annual_depreicable_capital
[ndarray] Depreciable capital by year.

Returns

annual_charge
[ndarray] Charge by year.

pyH2A.Discounted_Cash_Flow.**discounted_cash_flow_function**(*inp*, *values*, *parameters*, *attribute='h2_cost'*, *plugin=None*, *plugin_attr=None*)

Wrapper function for `Discounted_Cash_Flow`, substituting provided values at specified parameter positions and returning desired attribute of `Discounted_Cash_Flow` object.

Parameters

inp
[dict or str] Dictionary containing input information. If *inp* is a file path, the provided file is converted to a dictionary using `convert_input_to_dictionary`.

values
[ndarray] 1D (in case of one parameter) or 2D array (in case of multiple parameters) containing the values which are to be used.

parameters
[ndarray] 1D or 2D array containing the parameter specifications (location within inp); Format: [top_key, middle_key, bottom_key].

attribute
[str, optional] Desired attribute of `Discounted_Cash_Flow` object, which should be returned. If the attribute is *plugs*, the *.plugs* dictionary attribute is accessed, which contains information of all used plugins (see *plugin* and *plugin_attr*). Defaults to *h2_cost*.

plugin
[str, optional] If *attribute* is set to *plugs*, a *plugin* has to be specified, which should be accessed. Furthermore, a corresponding attribute of the plugin needs to be provided, see *plugin_attr*.

plugin_attr
[str, optional] If *attribute* is set to *plugs*, *plugin_attr* controls which attribute of the specified *plugin* is accessed.

Returns

results
[ndarray] For each value (1D array) or set of values (2D array), the values are substituted in inp, Discounted_Cash_Flow (dcf) is executed and the dcf object is generated. Then, the requested attribute is stored in results, which is finally returned.

pyH2A.Discounted_Cash_Flow.**get_idx**(*diagonal_number*, *axis0*, *axis1*)

Calculation of index for MACRS calculation. Uses `lru_cache` for repeated calculations.

pyH2A.Discounted_Cash_Flow.**numpy_npv**(*rate*, *values*)

Calculation of net present value.

2.4 Default Settings

```
# Workflow

Name | Type | Position | Description
--- | --- | --- | ---
Production_Scaling_Plugin | plugin | 1 | Computes plant output and scaling factors (if␣
↪scaling is requested)
production_scaling | function | 2 | core function to process yearly plant output
Capital_Cost_Plugin | plugin | 3 | Calculation of direct, indirect and non-depreciable␣
↪capital costs
initial_equity_depreciable_capital | function | 4 | core function to process depreciable␣
↪capital costs
non_depreciable_capital_costs | function | 5 | core function to process non-depreciable␣
↪capital costs
Replacement_Plugin | plugin | 6 | Calculation of replacement costs
replacement_costs | function | 7 | core function to process replacement costs
Fixed_Operating_Cost_Plugin | plugin | 8 |Calculation of fixed operating costs
fixed_operating_costs | function | 9 | core function to process fixed operating costs
Variable_Operating_Cost_Plugin | plugin | 10 | Calculation of variable operating costs,␣
↪including utilities
variable_operating_costs | function | 11 | core function to process variable operating␣
↪costs

# Financial Input Values

Name | Full Name | Value
--- | --- | ---
ref year | Reference year | 2016
startup year | Assumed start-up year | 2020
basis year | Basis year | 2016
current year capital costs | Current year for capital costs | 2016
startup time | Start-up Time (years) | 1
plant life | Plant life (years) | 20
depreciation length | Depreciation Schedule Length (years) | 20
depreciation type | Depreciation Type | MACRS
equity | % Equity Financing | 40%
interest | Interest rate on debt (%) | 3.7%
debt | Debt period | Constant
```

(continues on next page)

(continued from previous page)

```
startup cost fixed | % of Fixed Operating Costs During Start-up | 100%
startup revenues | % of Revenues During Start-up | 75%
startup cost variable | % of Variable Operating Costs During Start-up | 75%
decommissioning | Decommissioning costs (% of depreciable capital investment) | 10%
salvage | Salvage value (% of total capital investment) | 10%
inflation | Inflation rate (%) | 1.9%
irr | After-tax Real IRR (%) | 8.0%
state tax | State Taxes (%) | 6.0%
federal tax | Federal Taxes (%) | 21.0%
working capital | Working Capital (% of yearly change in operating costs) | 15.0%
```

CHAPTER

THREE

UTILITIES

3.1 Energy_Conversion

Classes:

Energy(value, unit)	Energy class to convert between different energy units.

Functions:

J(value)	Converts J to J
Jmol(value)	Converts J/mol to J
eV(value)	Converts eV to J
kJmol(value)	Converts kJ/mol to J
kWh(value)	Converts kWh to J
kcalmol(value)	Converts kcal/mol to J
nm(value)	Converts nm to J

`class` pyH2A.Utilities.Energy_Conversion.**Energy**(*value*, *unit*)

Energy class to convert between different energy units.

Parameters

value
[float] Input energy value.

unit
[function] Unit name which corresponds to the one of the functions defined outside of the class. This function is used to convert the input energy value to Joule.

Notes

Input value in either nm, eV, kcal/mol, J/mol, kJ/mol, kWh or J. Available units: J, eV, nm, kcal/mol, J/mol, kWh, kJ/mol. Once an Energy class object has been generated, the energy value in the desired unit can be retrieved using the appropriate class attribute.

Methods:

convert_J_to_Jmol()	Convert J to J/mol
convert_J_to_eV()	Convert J to eV
convert_J_to_kJmol()	Convert to J to kJ/mol
convert_J_to_kWh()	Convert J to kWh
convert_J_to_kcalmol()	Convert J to kcal/mol
convert_J_to_nm()	Convert J to nm

convert_J_to_Jmol()
Convert J to J/mol

convert_J_to_eV()
Convert J to eV

convert_J_to_kJmol()
Convert to J to kJ/mol

convert_J_to_kWh()
Convert J to kWh

convert_J_to_kcalmol()
Convert J to kcal/mol

convert_J_to_nm()
Convert J to nm

pyH2A.Utilities.Energy_Conversion.**J**(*value*)
Converts J to J

pyH2A.Utilities.Energy_Conversion.**Jmol**(*value*)
Converts J/mol to J

pyH2A.Utilities.Energy_Conversion.**eV**(*value*)
Converts eV to J

pyH2A.Utilities.Energy_Conversion.**kJmol**(*value*)
Converts kJ/mol to J

pyH2A.Utilities.Energy_Conversion.**kWh**(*value*)
Converts kWh to J

pyH2A.Utilities.Energy_Conversion.**kcalmol**(*value*)
Converts kcal/mol to J

pyH2A.Utilities.Energy_Conversion.**nm**(*value*)
Converts nm to J

3.2 find_nearest

Functions:

find_nearest(array, values)	Find value(s) in *array* that are nearest to *values*.

pyH2A.Utilities.find_nearest.**find_nearest**(*array*, *values*)

Find value(s) in *array* that are nearest to *values*.

Parameters

array: ndarray
Array to be searched. If *array* has more than one dimension, only the first column is used.

values
[float, ndarray or list] Single float, ndarray or list with values for which the nearest entries in *array* should be found.

Returns

hits
[list] List of indices of closest values in *array*.

3.3 input_modification

Functions:

check_for_meta_module(key)	Checks if *key* is a meta module that is to be executed.
convert_dict_to_kwargs_dict(dictionary[, ...])	Converting dictionary generated by `convert_file_to_dictionary()` to a dictionary that can be used to provide keyword arguments.
convert_file_to_dictionary(file)	Convert provided text file into dictionary.
convert_input_to_dictionary(file[, default, ...])	Reads provided input file (file) and default file, converting both to dictionaries.
execute_plugin(plugin_name, plugs_dict[, ...])	Executing module.
file_import(file_name[, mode, return_path])	Importing package file or file at arbitrary path and returning typing.TextIO instance.
get_by_path(root, items)	Access a nested object in *root* by item sequence.
import_plugin(plugin_name, plugin_module)	Importing module.
insert(class_object, top_key, middle_key, ...)	Insert function used in plugins.
merge(a, b[, path, update])	Deep merge two dictionaries, with b having priority over a
num(s)	Converting string to either an int, float, or, if neither is possible, returning the string.
parse_parameter(key[, delimiter])	Provided *key* is split at delimiter(s) and returned as cleaned array
parse_parameter_to_array(key[, delimiter, ...])	parse_parameter() is applied to *key* string and result is converted to num and returned in ndarray
process_cell(dictionary, top_key, key, ...)	Processing of a single cell at dictionary[top_key][key][bottom_key]
process_input(dictionary, top_key, key, ...)	Processing of input at dictionary[top_key][key][bottom_key].
process_path(dictionary, path, top_key, key, ...)	Processing provided path.
process_table(dictionary, top_key, bottom_key)	Looping through all keys in dictionary[top_key] and applying process_input to dictionary[top_key][key][bottom_key].
read_textfile(file_name, delimiter[, mode])	Wrapper for genfromtxt with lru_cache for repeated reads of the same file.
reverse_parameter_to_string(parameter)	Reverts processed parameter list to string.
set_by_path(root, items, value[, value_type])	Set a value in a nested object in root by item sequence.
sum_all_tables(dictionary, table_group, ...)	Applies `sum_table()` to all dictionary entries with a key that contains *table_group*.
sum_table(dictionary, top_key, bottom_key[, ...])	For the provided *dictionary*, all entries in dictionary[top_key] are processed using `process_input()` (positions: top_key > key > bottom key) and summed.

pyH2A.Utilities.input_modification.**check_for_meta_module**(*key*)

Checks if *key* is a meta module that is to be executed.

Notes

Meta module is identified by checking if *key* contains the substring 'Analysis' and does not contain any of the substrings in *exceptions*.

pyH2A.Utilities.input_modification.**convert_dict_to_kwargs_dict**(*dictionary*, *middle_key='Value'*)

Converting dictionary generated by `convert_file_to_dictionary()` to a dictionary that can be used to provide keyword arguments.

Parameters

dictionary
[dict] Dictionary to be converted.

middle_key
[str, optional] Middle key which is present in input dictionary. This key is removed in the process.

Returns

output
[dict] Dictionary suitable to provide keyword arguments.

pyH2A.Utilities.input_modification.**convert_file_to_dictionary**(*file*)

Convert provided text file into dictionary. Text file has to follow GitHub flavoured Markdown style.

Parameters

file
[typing.TextIO] typing.TextIO instance of file to be converted.

Returns

inp
[dict] Dictionary containing converted data from file.

Notes

Table format:

Table A name

First | Second | …

— | — | —

Entry A | value 1 | …

Entry B | value 2 | …

Table B name

…

The table name is used as *top_key*, the entries within the first column are used as *middle_key* and the names of the other columns are used as *bottom key*. E.g. {'Table A name': {'Entry A' : {'Second': 'value 1'}}}

pyH2A.Utilities.input_modification.**convert_input_to_dictionary**(*file*, *default='pyH2A.Config~Defaults.md'*, *merge_default=True*)

Reads provided input file (file) and default file, converting both to dictionaries. The dictionaries are merged, with the input file having priority.

Parameters

file
[str] Path to input file.

default
[str, optional] Path to default file.

merge_default
[bool] Flag to control if input is merged with default file.

Returns

inp
[dict] Input dictionary.

pyH2A.Utilities.input_modification.**execute_plugin**(*plugin_name*, *plugs_dict*, *plugin_module=True*, *nested_dictionary=False*, ***kwargs*)

Executing module.

Parameters

plugin_name
[str] Name of module.

plugs_dict
[dict] Dictionary to store class object generated from module.

plugin_module
[bool, optional] Flag to differentiate between plugins and analysis modules. If *True*, module is imported from *Plugins.* directory. If *False*, it is imported from *Analysis.* directory.

nested_dictioanry
[bool, optional] If *True*, a sub dictionary is created in *plugs_dict*, where the class object is stored.

****kwargs:**
Keyword arguments passed to class within module.

Returns

plugin_object:
Class objected generated from module.

Notes

Module *plugin_name* is imported. It is assumed that the module contains a class with the same name as *plugin_name*. An instance of this class is created using ***kwargs*. The class object is then stored in *plugs_dict*.

pyH2A.Utilities.input_modification.**file_import**(*file_name*, *mode='rb'*, *return_path=False*)

Importing package file or file at arbitrary path and returning typing.TextIO instance.

Parameters

file_name
[str] Path to file to be read. Can be either a regular path or a path of the form *package.subdirectory~file_name* to refer to a file in the pyH2A installation.

mode
[str] Mode for file read. Can be either *r* or *rb*. In case of *r*, a *typing.TextIO* instance is returned. In case of *rb* a *typing.BinaryIO* instance is returned.

Returns

output
[typing.BinaryIO or typing.TextIO instance] Whether a *typing.BinaryIO* or *typing.TextIO* is returned depends on *mode*.

pyH2A.Utilities.input_modification.**get_by_path**(*root*, *items*)

Access a nested object in *root* by item sequence.

pyH2A.Utilities.input_modification.**import_plugin**(*plugin_name*, *plugin_module*)

Importing module.

Parameters

plugin_name
[str] Name of module.

plugin_module
[bool, optional] Flag to differentiate between plugins and analysis modules. If *True*, module is imported from *Plugins.* directory. If *False*, it is imported from *Analysis.* directory.

Returns

plugin_class:
Class from imported module.

Notes

Module *plugin_name* is imported. It is assumed that the module contains a class with the same name as *plugin_name*

pyH2A.Utilities.input_modification.**insert**(*class_object*, *top_key*, *middle_key*, *bottom_key*, *value*, *name*, *print_info=True*, *add_processed=True*, *insert_path=True*)

Insert function used in plugins.

Parameters

class_object: Discounted_Cash_Flow object
Dicounted_Cash_Flow object with .inp attribute, which is modified.

top_key
[str] Top key.

middle_key
[str] Middle key.

bottom_key
[str] Bottom key.

Value
[int, float, str or ndarray] Value inserted at top_key > middle_key > bottom_key position.

name
[str] Name of plugin performing insertion.

print_info
[bool, optional] Flag to control if information on action of `insert()` is printed.

add_processed
[bool, optional] Flag to control if 'Processed' key is added.

insert_path
[bool, optional] Flog to control if 'Path' key is added.

Notes

inp attribute of *class_object* is modified by inserting *value* at the position defined by *top_key* > *middle_key* > *bottom_key*. If there already is a value at this position, it will be overwritten. In this case, the 'Path' entry will be set to 'None' to avoid issues if value at this position already existed and had a path specified. If there is not already a value at this position, it will be created *name* is the name of plugin using *insert* for insertion. If *print_info* is True, action of *insert* will be printed.

pyH2A.Utilities.input_modification.**merge**(*a*, *b*, *path=None*, *update=True*)

Deep merge two dictionaries, with b having priority over a

pyH2A.Utilities.input_modification.**num**(*s*)

Converting string to either an int, float, or, if neither is possible, returning the string.

Parameters

s
[str] String to be converted to a number.

Returns

num
[int, float or str] String converted to int, float or str.

Notes

Input strings can contain commas as thousands seperators, which will be removed if the string is otherwise a valid number (float or int). If the input string ends with a "%" sign, it will be converted to a number divided by 100.

pyH2A.Utilities.input_modification.**parse_parameter**(*key*, *delimiter='>'*)

Provided *key* is split at delimiter(s) and returned as cleaned array

pyH2A.Utilities.input_modification.**parse_parameter_to_array**(*key*, *delimiter='>'*, *dictionary=None*, *top_key=None*, *middle_key=None*, *bottom_key=None*, *special_values=[]*, *path=None*)

parse_parameter() is applied to *key* string and result is converted to num and returned in ndarray

Parameters

key
[str] String convert to array.

delimiter
[str, optional] Delimiter used in string.

dictionary
[dict, optional] Dictionary used for lookup.

top_key
[str, optional] Top key for lookup.

middle_key
[str, optional] Middle key for lookup.

bottom_key
[str, optional] Bottom key for lookup.

special_values
[list, optional] If *key* contains an element of *special_values*, the value at *path* is retrieved instead of using the actual value of *key*.

path
[str, optional] Path for lookup in case *special_values* is triggered.

Returns

array
[ndarray] Output as array.

pyH2A.Utilities.input_modification.**process_cell**(*dictionary*, *top_key*, *key*, *bottom_key*, *cell=None*, *print_processing_warning=True*)

Processing of a single cell at dictionary[top_key][key][bottom_key]

Parameters

dictionary
[dict] Dictionary within which function operates.

top_key
[str] Top key.

key
[str] Middle key.

bottom_key
[str] Bottom key.

cell
[int, float, str or None] Cell entry.

print_processing_warning
[bool] Flag to control if a warning is printed when an unprocessed value is being used.

Notes

If cell contains only a number, the contents of that cell are returned. If cell contains a string, but that string is not a path (indicated by absence of ">" symbol), 1 is returned If cell contains a string which is potentially a path, it is processed: Contents of the cell are split at ";" delimiter, separating multiple potential paths. For each potential path, process_path() is applied. The retrieved target value(s) are multiplied and returned. Since value is initated to 1, if none of the paths are valid, simply 1 is returned.

pyH2A.Utilities.input_modification.**process_input**(*dictionary*, *top_key*, *key*, *bottom_key*, *path_key='Path'*, *add_processed=True*)

Processing of input at dictionary[top_key][key][bottom_key].

Parameters

dictionary
[dict] Dictionary within which function operates.

top_key
[str] Top key.

key
[str] Middle key.

bottom_key
[str] Bottom key.

path_key
[str, optional] Key used for path column. Defaults to 'Path'.

add_processed
[bool, optional] Flag to control if *Processed* key is added

Notes

Action: if there is an entry at dictionary[top_key][key][path_key], process_input() applies process_cell() to dictionary[top_key][key][bottom_key] as well as dictionary[top_key][key][path_key] and multiplies them. The resulting value is returned and placed into dictionary[top_key][key][bottom_key]

Detailed Description:

First, it is checked if that input has already been processed by looking for the "Processed" key. If this is the case, the input is simply returned. If it has not already been processed, it is checked if the input is a string which could not be a path (not containing ">"). In this case the string is simply returned and "Processed" is added.

If neither condition is met, process_cell() is applied. It is then attempted to retrieve dictionary[top_key][key][path_key]. If this entry cannot be retrieved, the process_cell() value of the input is returned,

If this entry can be retrieved, process_cell() is applied to it and the resulting Value is multiplied by the original process_cell() value of the input, updating value.

If the obtained value differs from the original entry, the obtained value is inserted at dictionary[top_key][key][bottom_key] and the original entry is stored in dictionary[top_key][key][former_bottom_key]

At the end, "Processed" is added.

pyH2A.Utilities.input_modification.**process_path**(*dictionary*, *path*, *top_key*, *key*, *bottom_key*, *print_processing_warning=True*)

Processing provided path. Checks are performed to see if path is valid.

Parameters

dictionary
[dict] Dictionary within which function operates.

path
[str] Path.

top_key
[str] Top key.

key
[str] Middle key.

bottom_key
[str] Bottom key.

print_processing_warning
[bool] Flag to control if a warning is printed when an unprocessed value is being used.

Notes

If provided path contains no ">" symbols, it is not a path and 1 is returned If provided path contains only one ">" symbol, it is not a valid path. A warning is printed and 1 is returned. If provided path contains two ">" symbols, it is potentially a valid path. It is then attempted to retrieve target value. If retrieval attempt is unsuccessful, a warning is printed and 1 is returned. If the path is valid, the target value is retrieved: If the rerieved target value comes from an unprocessed key, a warning is printed. If the retrieved target value is non-numerical, a warning is printed and 1 is returned. If the retrieved target value is numerical, it is returned.

pyH2A.Utilities.input_modification.**process_table**(*dictionary*, *top_key*, *bottom_key*, *path_key='Path'*)

Looping through all keys in dictionary[top_key] and applying process_input to dictionary[top_key][key][bottom_key].

Parameters

dictionary
[dict] Dictionary within which function operates.

top_key
[str] Top key.

bottom_key
[str, ndarray or list] Bottom key(s).

path_key
[str or ndarray, optional] Key(s) used for path column(s). Defaults to 'Path'.

Notes

bottom_key can also be an array of keys, all of which are processed (in this case, path_key has to be an array of equal length).

pyH2A.Utilities.input_modification.**read_textfile**(*file_name*, *delimiter*, *mode='rb'*, ***kwargs*)

Wrapper for genfromtxt with lru_cache for repeated reads of the same file.

Parameters

file_name
[str] Path to file.

delimiter
[str] Delimiter used in file.

****kwargs:**
Keyword arguments passed to `numpy.genfromtxt()`.

Returns

data
[ndarray] Array containing read data.

pyH2A.Utilities.input_modification.**reverse_parameter_to_string**(*parameter*)

Reverts processed parameter list to string.

pyH2A.Utilities.input_modification.**set_by_path**(*root*, *items*, *value*, *value_type='value'*)

Set a value in a nested object in root by item sequence.

Notes

Existing value is either multiplied by provided one (value_type = factor) or is replaced by provided one. In-place replacement, should only be used on deep copy of self.inp dictionary

pyH2A.Utilities.input_modification.**sum_all_tables**(*dictionary*, *table_group*, *bottom_key*, *insert_total=False*, *class_object=None*, *middle_key_insertion='Summed Total'*, *bottom_key_insertion='Value'*, *print_info=True*, *path_key='Path'*, *return_contributions=False*)

Applies `sum_table()` to all dictionary entries with a key that contains *table_group*. Resulting `sum_table()` values are summed to return total.

Parameters

dictionary
[dict] Dictionary within which function operates.

table_group
[str] String to identify table group. If a dictionary key contains the *table_group* substring it is part of the table group.

bottom_key
[str, ndarray or list] Bottom key.

insert_total
[bool, optional] If *insert_total* is True, the total of each table is inserted in the respective table.

class_object
[Discounted_Cash_Flow object] Discounted_Cash_Flow object whose .inp attribute is modified.

middle_key_insertion
[str, optional] Middle key used for insertion of total.

bottom_key_insertion
[str, optional] Bottom key used for insertion of total.

print_info
[bool, optional] Flag to control if information on action of `insert()` is printed.

path_key
[str, optional] Key used for path column. Defaults to 'Path'.

return_contributions
[bool, optional] Flag to control if a dictionary with contributions breakdown (for use in cost `Cost_Contributions_Analysis` module) is returned.

Notes

If *insert_total* is true, the `sum_table()` value for a given key is inserted in class_object.inp at key > middle_key_insertion > bottom_key_insertion.

The contributions of each table in table_group are stored in *contributions* dictionary, which is returned if *return_contributions* is set to True. Dictionary is structured so that it can be provided to "Cost_Contributions_Analysis" class to generate a cost breakdown plot.

pyH2A.Utilities.input_modification.**sum_table**(*dictionary*, *top_key*, *bottom_key*, *path_key='Path'*)

For the provided *dictionary*, all entries in dictionary[top_key] are processed using `process_input()` (positions: top_key > key > bottom key) and summed.

Parameters

dictionary
[dict] Dictionary within which function operates.

top_key
[str] Top key.

bottom_key
[str, ndarray or list] Bottom key.

path_key
[str, optional] Key used for path column. Defaults to 'Path'.

3.4 output_utilities

Classes:

Figure_Lean(name, directory[, ...])	Wrapper class for figures.
MathTextSciFormatter([fmt])	Formatter for scientific notation in MathText.

Functions:

bottom_offset(self, bboxes, bboxes2)	Bottom offset for cost contribution plot labels.
dynamic_value_formatting(value[, cutoff])	Dynamic formatiing of value to string.
format_scientific(value)	Converts value to string with scientfic (10**x) notation
insert_image(path, x, y, zoom, ax)	Insert image into plot.
make_bold(string)	Convert provided string to a string which is formatted to be bold.
millify(n[, dollar_sign])	Converts n to a string with shorthand notation for thousand-steps
set_font(font_family, font, font_size)	Set font for plot.

class pyH2A.Utilities.output_utilities.**Figure_Lean**(*name*, *directory*, *provided_figure_and_axis=None*, *show=False*, *save=False*, *pdf=True*, *dpi=300*, *transparent=False*, *nrows=1*, *ncols=1*, *fig_width=6.4*, *fig_height=4.8*, *left=0.125*, *right=0.9*, *top=0.88*, *bottom=0.11*, *wspace=0.2*, *hspace=0.2*, *font_family='sans-serif'*, *font='Arial'*, *font_size=12*, *sharex=False*, *input_file_name=None*, *append_file_name=True*)

Wrapper class for figures.

Parameters

name
[str] Name of figure, used for saving.

directory
[str] Path to directory where figure is to be saved.

provided_figure_and_axis
[tuple or None, optional] Tuple of matplotlib.fig and matplotlib.ax objects. If `None` new *fig* and *ax* objects are generated.

show
[bool, optional] If True, figure is shown.

save
[bool, optional] If True, figure is saved.

pdf
[bool, optional] If True, figure is saved as a PDF file. If False, it is saved as a PNG file.

dpi
[int, optional] Dots per inch resolution of figure.

transparent
[bool, optional] Flag to control if background of figure is transparent or not.

nrows
[int, optional] Number of rows for subplots.

ncols
[int, optional] Number of columns for suplots.

fig_width
[float, optional] Width of figure in inches.

fig_height
[float, optional] Height of figure in inches.

left
[float, optional] Left edge of plot in axis fraction coordinates.

right
[float, optional] Right edge of plot in axis fraction coordinates.

top
[float, optional] Top edge of plot in axis fraction coordinates.

bottom
[float, optional] Bottom edge of plot in axis fraction coordinates.

wspace
[float, optional] Vertical white space between subplots.

hspace
[float, optional] Horizontal white space between subplots.

font_family
[str, optional] Font family, either 'serif' or 'sans-serif'.

font
[str, optional] Name of font.

font_size
[float, optional] Font size.

sharex
[bool, optional] Flag to control if x axis is shared between subplots.

input_file_name
[str, optional] Name of input file.

append_file_name ; bool, optional
Flag to control if *input_file_name* is appended to file name of figure.

Notes

Provided figure is shown and/or saved in provided directory with given name by running *Figure_Lean.execute()*.

Methods:

execute()	Running *self.execute()* executes desired *show* and *save* options.
save_figure(pdf, dpi, transparent)	Saving figure in target dictionary with specified parameters.

execute()

Running *self.execute()* executes desired *show* and *save* options.

save_figure(*pdf*, *dpi*, *transparent*)

Saving figure in target dictionary with specified parameters.

class pyH2A.Utilities.output_utilities.**MathTextSciFormatter**(*fmt='%1.1e'*)

Formatter for scientific notation in MathText.

Methods

__call__:	Call method.
fix_minus:	Fixing minus.
format_data:	Format data method.
format_data_short:	Format data shortened method.
format_ticks:	Format ticks methods.

pyH2A.Utilities.output_utilities.**bottom_offset**(*self*, *bboxes*, *bboxes2*)

Bottom offset for cost contribution plot labels.

pyH2A.Utilities.output_utilities.**dynamic_value_formatting**(*value*, *cutoff=6*)

Dynamic formatiing of value to string.

Parameters

cutoff
[int, optional] Cutoff value for string length. Below cutoff value a string is shown without special formatting.

Notes

If value is an int (or a float that can be represented as an int) and its length as a string is less than the *cutoff value*, it will be printed as such. If its length as a string is more than the cutoff value, it will be either printed using the *millify* function (if the value is larger than 1), or using the *format_scientific* function (if the value is smaller than 1).

pyH2A.Utilities.output_utilities.**format_scientific**(*value*)

Converts value to string with scientfic (10**x) notation

pyH2A.Utilities.output_utilities.**insert_image**(*path*, *x*, *y*, *zoom*, *ax*)

Insert image into plot.

Parameters

path
[str] Path to image to be inserted.

x
[float] x axis coordinate of image in axis fraction coordinates.

y
[float] y axis coordinate of image in axis fraction coordinates.

ax
[matplotlib.ax] matplotlib.ax object into which image is inserted.

pyH2A.Utilities.output_utilities.**make_bold**(*string*)

Convert provided string to a string which is formatted to be bold. "%" signs cannot be rendered bold with this approach and are hence returned normal

pyH2A.Utilities.output_utilities.**millify**(*n*, *dollar_sign=True*)

Converts n to a string with shorthand notation for thousand-steps

pyH2A.Utilities.output_utilities.**set_font**(*font_family*, *font*, *font_size*)

Set font for plot.

Parameters

font_family
[str] Font family, either 'serif' or 'sans-serif'.

font
[str] Name of font.

font_size
[float] Font size.

3.5 plugin_input_output_processing

Classes:

Generate_Template_Input_File(...[, origin, ...])	Generate input file template from a minimal input file.
Template_File(inp)	Generate input file from inp dictionary.

Functions:

convert_inp_to_requirements(dictionary[, path])	Convert inp dictionary structure to requirements dictionary structure.
extract_input_output_from_docstring(target, ...)	Convert docstring to structured dictionary.
is_parameter_or_output(line[, ...])	Detection of parameters and output values in line based on presence of more than *spaces_cuttoff* spaces (tabs are converted to four spaces).
process_single_line(line, output_dict, ...)	Process single line to extract parameter/output information and comments

class pyH2A.Utilities.plugin_input_output_processing.**Generate_Template_Input_File**(*input_file_stub*, *output_file*, *origin=False*, *comment=False*)

Generate input file template from a minimal input file.

Parameters

input_file_stub
[str] Path to input file containing workflow and analysis specifications.

output_file
[str] Path to file where input template is to be written.

origin
[bool, optional] Include origin of each requested input parameter in input template file ("requested by" information).

comment
[bool, optional] Include comments for each requested input parameter (additional information on parameter).

Returns

Template
[object] Template object which contains information on requirements and output. Input template is written to specified output file.

Methods:

check_parameters(data, output)	Check if needed parameter is in output.
convert_requirements_to_inp([insert_origin, ...])	Convert dictionary of requirements to formatted self.inp
generate_requirements()	Generate dictionary with input requirements.
get_analysis_modules(post_workflow_position)	Get analysis modules from input stub.
get_docstring_data(target_name, target_type)	Get parameter requirements and outputs from docstrings.

check_parameters(*data*, *output*)

Check if needed parameter is in output.

convert_requirements_to_inp(*insert_origin=False*, *insert_comment=False*)

Convert dictionary of requirements to formatted self.inp

generate_requirements()

Generate dictionary with input requirements.

get_analysis_modules(*post_workflow_position*)

Get analysis modules from input stub.

get_docstring_data(*target_name*, *target_type*)

Get parameter requirements and outputs from docstrings.

class pyH2A.Utilities.plugin_input_output_processing.**Template_File**(*inp*)

Generate input file from inp dictionary.

Parameters

inp

[dict] Dictionary containing information on requested input (generated by *Generate_Template_Input_File*)

Returns

Template_File

[object] Object containing formatted string for output file.

Methods:

convert_column_names_to_string(names)	Convert list of names to markdown style table string.
convert_inp_to_string()	Convert inp to string.
get_column_names(dictionary)	Get names of table columns.
get_row_entries(columns, column_names, ...)	Get entries for each row of table.
get_single_row(column_names, column, dictionary)	Get entries for a single row.
write_template_file(file_name)	Write output string to file.

convert_column_names_to_string(*names*)

Convert list of names to markdown style table string.

convert_inp_to_string()

Convert inp to string.

get_column_names(*dictionary*)

Get names of table columns.

get_row_entries(*columns*, *column_names*, *dictionary*)

Get entries for each row of table.

get_single_row(*column_names*, *column*, *dictionary*)

Get entries for a single row.

write_template_file(*file_name*)

Write output string to file.

pyH2A.Utilities.plugin_input_output_processing.**convert_inp_to_requirements**(*dictionary*, *path=None*)

Convert inp dictionary structure to requirements dictionary structure.

pyH2A.Utilities.plugin_input_output_processing.**extract_input_output_from_docstring**(*target*, ***kwargs*)

Convert docstring to structured dictionary.

pyH2A.Utilities.plugin_input_output_processing.**is_parameter_or_output**(*line*, *spaces_for_tab=4*, *spaces_cutoff=5*)

Detection of parameters and output values in line based on presence of more than *spaces_cuttoff* spaces (tabs are converted to four spaces).

pyH2A.Utilities.plugin_input_output_processing.**process_single_line**(*line*, *output_dict*, *origin*, *variable_string*, ***kwargs*)

Process single line to extract parameter/output information and comments

CHAPTER

FOUR

PLUGINS

Plugins allow for modelling of different hydrogen production pathways. They process information and feed it back into the discounted cashflow calculation to model different behaviour (e.g. energy generation, storage, conversion, cost processing etc.)

4.1 Plugin Guide

pyH2A follows the open-closed principle, meaning that new plugins can be interfaced with pyH2A without modification of the source code. To work with pyH2A, plugins should follow certain design principles.

4.1.1 Structure

Plugins are single `.py` files, which contain a class with the same name as the filename (this shared name should include the term `Plugin`). This class is instantiated during pyH2A runtime. Instantiation requires two arguments: `dcf`, which is discounted cash flow object (generated during pyH2A runtime) and `print_info`, which is a flag to control printing of additional runtime information (this flag is passed to *insert()*) The file may also contain other classes and functions, which serve the central class.

The overall idea is that during instantiation, the class reads information from `dcf.inp` (the dictionary generated from the input file), processes the information and inserts new information into `dcf.inp`. `dcf.inp` is the medium of information exchange between plugins and by inserting information there, the results of the plugin affect the outcome of the discounted cashflow analysis.

4.1.2 Example

```
from pyH2A.Utilities.input_modification import insert, process_table

class Example_Plugin:
        '''Docstring header.

        Parameters
        ----------
        Table > Row > Value : float
                Example input read from input file by plugin.

        Returns
        -------
        Other Table > Row > Value : float
```

(continues on next page)

(continued from previous page)

```
                Output generated by plugin which is stored in dcf.inp.
        '''

        def __init__(self, dcf, print_info):
                process_table(dcf.inp, 'Table', 'Value') # 'Value' column of 'Table' is␣
↪processed

                self.method(dcf)

                insert(dcf, 'Other Table', 'Row', 'Value', self.attribute,
                                __name__, print_info = print_info)

        def method(self, dcf):
                '''Calculation performed by plugin. In this case the input information is␣
↪only
                read and stored in an attribute.
                '''

                self.attribute = dcf.inp['Table']['Row']['Value']
```

In this example, the plugin reads information from `Table > Row > Value`, processes it by applying the *process_table()* function (which ensures that references are resolved) and it runs `self.method(dcf)`, during which it stores the input information in an attribute `self.attribute`. Finally, `self.attribute` is inserted into `dcf.inp` in a new location: `Other Table > Row > Value`.

With this pattern, plugins can process information from other plugins which ran before it in the `Workflow` (determined by the `Workflow` position) and plugins running after it can use the information inserted in to `dcf.inp`.

It is important to include the `numpydoc` formatted docstring, since it is used to generate the the input file template.

4.2 Capital_Cost_Plugin

Classes:

Capital_Cost_Plugin(dcf, print_info)	**Parameters**

class pyH2A.Plugins.Capital_Cost_Plugin.**Capital_Cost_Plugin**(*dcf*, *print_info*)

Parameters

[...] Direct Capital Cost [...] >> Value
[float] `sum_all_tables()` is used.

[...] Indirect Capital Cost [...] >> Value
[float] `sum_all_tables()` is used.

Non-Depreciable Capital Costs > Cost of land ($ per acre) > Value
[float] Cost of land in $ per acre, `process_table()` is used.

Non-Depreciable Capital Costs > Land required (acres) > Value
[float] Total land are required in acres, `process_table()` is used.

[...] Other Non-Depreciable Capital Cost [...] >> Value
[float] `sum_all_tables()` is used.

Returns

[...] Direct Capital Cost [...] > Summed Total > Value
[float] Summed total for each individual table in "Direct Capital Cost" group.

[...] Indirect Capital Cost [...] > Summed Total > Value
[float] Summed total for each individual table in "Indirect Capital Cost" group.

[...] Other Non-Depreciable Capital Cost [...] > Summed Total > Value
[float] Summed total for each individual table in "Other Non-Depreciable Capital Cost" group.

Direct Capital Costs > Total > Value
[float] Total direct capital costs.

Direct Capital Costs > Inflated > Value
[float] Total direct capital costs multiplied by combined inflator.

Indirect Capital Costs > Total > Value
[float] Total indirect capital costs.

Indirect Capital Costs > Inflated > Value
[float] Total indirect capital costs multiplied by combined inflator.

Non-Depreciable Capital Costs > Total > Value
[float] Total non-depreciable capital costs.

Non-Depreciable Capital Costs > Inflated > Value
[float] Total non-depreciable capital costs multiplied by combined inflator.

Depreciable Capital Costs > Total > Value
[float] Sum of direct and indirect capital costs.

Depreciable Capital Costs > Inflated > Value
[float] Sum of direct and indirect capital costs multiplied by combined inflator.

Total Capital Costs > Total > Value
[float] Sum of depreciable and non-depreciable capital costs.

Total Capital Costs > Inflated > Value
[float] Sum of depreicable and non-depreciable capital costs multiplied by combined inflator.

['Capital_Cost_Plugin'].direct_contributions
[dict] Attribute containing cost contributions for "Direct Capital Cost" group.

Methods:

direct_capital_costs(dcf, print_info)	Calculation of direct capital costs by applying `sum_all_tables()` to "Direct Capital Cost" group.
indirect_capital_costs(dcf, print_info)	Calculation of indirect capital costs by applying `sum_all_tables()` to "Indirect Capital Cost" group.
non_depreciable_capital_costs(dcf, print_info)	Calculation of non-depreciable capital costs by calculating cost of land and applying `sum_all_tables()` to "Other Non-Depreciable Capital Cost" group.

`direct_capital_costs`(*dcf*, *print_info*)

Calculation of direct capital costs by applying `sum_all_tables()` to "Direct Capital Cost" group.

indirect_capital_costs(*dcf*, *print_info*)

Calculation of indirect capital costs by applying `sum_all_tables()` to "Indirect Capital Cost" group.

non_depreciable_capital_costs(*dcf*, *print_info*)

Calculation of non-depreciable capital costs by calculating cost of land and applying `sum_all_tables()` to "Other Non-Depreciable Capital Cost" group.

4.3 Catalyst_Separation_Plugin

Classes:

Catalyst_Separation_Plugin(dcf, print_info)	Calculation of cost for catalyst separation (e.g.

class pyH2A.Plugins.Catalyst_Separation_Plugin.**Catalyst_Separation_Plugin**(*dcf*, *print_info*)

Calculation of cost for catalyst separation (e.g. via nanofiltration).

Parameters

Water Volume > Volume (liters) > Value
[float] Total water volume in liters.

Catalyst > Lifetime (years) > Value
[float] Lifetime of catalysts in year before replacement is required.

Catalyst Separation > Filtration cost ($/m3) > Value
[float] Cost of filtration in $ per m3.

Returns

Other Variable Operating Cost - Catalyst Separation > Catalyst Separation (yearly cost) > Value
[float] Yearly cost of catalyst seperation.

Methods:

calculate_filtration_cost(dcf)	Yearly cost of water filtration to remove catalyst.
calculate_yearly_filtration_volume(dcf)	Calculation of water volume that has to be filtered per year.

calculate_filtration_cost(*dcf*)

Yearly cost of water filtration to remove catalyst.

calculate_yearly_filtration_volume(*dcf*)

Calculation of water volume that has to be filtered per year.

4.4 Fixed_Operating_Cost_Plugin

Classes:

Fixed_Operating_Cost_Plugin(dcf, print_info)	Calculation of yearly fixed operating costs.

class pyH2A.Plugins.Fixed_Operating_Cost_Plugin.**Fixed_Operating_Cost_Plugin**(*dcf*, *print_info*)

Calculation of yearly fixed operating costs.

Parameters

Fixed Operating Costs > staff > Value
[float] Number of staff, `process_table()` is used.

Fixed Operating Costs > hourly labor cost > Value
[float] Hourly labor cost of staff, `process_table()` is used.

[...] Other Fixed Operating Cost [...] >> Value
[float] Yearly other fixed operating costs, `sum_all_tables()` is used.

Returns

[...] Other Fixed Operating Cost [...] > Summed Total > Value
[float] Summed total for each individual table in "Other Fixed Operating Cost" group.

Fixed Operating Costs > Labor Cost - Uninflated > Value
[float] Yearly total labor cost.

Fixed Operating Costs > Labor Cost > Value
[float] Yearly total labor cost multiplied by labor inflator.

Fixed Operating Costs > Total > Value
[float] Sum of total yearly labor costs and yearly other fixed operating costs.

Methods:

labor_cost(dcf)	Calculation of yearly labor costs by multiplying number of staff times hourly labor cost.
other_cost(dcf, print_info)	Calculation of yearly other fixed operating costs by applying `sum_all_tables()` to "Other Fixed Operating Cost" group.

labor_cost(*dcf*)

Calculation of yearly labor costs by multiplying number of staff times hourly labor cost.

other_cost(*dcf*, *print_info*)

Calculation of yearly other fixed operating costs by applying `sum_all_tables()` to "Other Fixed Operating Cost" group.

4.5 Hourly_Irradiation_Plugin

Classes:

Hourly_Irradiation_Plugin(dcf, print_info)	Calculation of hourly and mean daily irradiation data with different module configurations.

Functions:

calculate_PV_power_ratio(file_name, ...)	Calculation based on Chang 2020, https://doi.org/10.1016/j.xcrp.2020.100209 SAT: horzontal single axis tracking DAT: dual axis tracking, no diffuse radiation
converter_function(string)	Converter function for datetime of hourly irradiation data.
import_Chang_data(file_name)	Import of Chang 2020 data, for debugging.
import_hourly_data(file_name)	Imports hourly irradiation data and location coordinates from the *.csv* format provided by: https://re.jrc.ec.europa.eu/pvg_tools/en/#TMY.

class pyH2A.Plugins.Hourly_Irradiation_Plugin.**Hourly_Irradiation_Plugin**(*dcf*, *print_info*)

Calculation of hourly and mean daily irradiation data with different module configurations.

Parameters

Hourly Irradiation > File > Value
[str] Path to a *.csv* file containing hourly irradiance data as provided by https://re.jrc.ec.europa.eu/pvg_tools/en/#TMY, `process_table()` is used.

Irradiance Area Parameters > Module Tilt (degrees) > Value
[float] Tilt of irradiated module in degrees.

Irradiance Area Parameters > Array Azimuth (degrees) > Value
[float] Azimuth angle of irradiated module in degrees.

Irradiance Area Parameters > Nominal Operating Temperature (Celsius) > Value
[float] Nominal operating temperature of irradiated module in degrees Celsius.

Irradiance Area Parameters > Mismatch Derating > Value
[float] Derating value due to mismatch (percentage or value between 0 and 1).

Irradiance Area Parameters > Dirt Derating > Value
[float] Derating value due to dirt buildup (percentage or value between 0 and 1).

Irradiance Area Parameters > Temperature Coefficient (per Celsius) > Value
[float] Performance decrease of irradiated module per degree Celsius increase.

Returns

Hourly Irradiation > No Tracking (kW) > Value
[ndarray] Hourly irradiation with no tracking per m2 in kW.

Hourly Irradiation > Horizontal Single Axis Tracking (kW) > Value
[ndarray] Hourly irradiation with single axis tracking per m2 in kW.

Hourly Irradiation > Two Axis Tracking (kW) > Value
[ndarray] Hourly irradiation with two axis tracking per m2 in kW.

Hourly Irradiation > Mean solar input (kWh/m2/day) > Value
[float] Mean solar input with no tracking in kWh/m2/day.

Hourly Irradiation > Mean solar input, single axis tracking (kWh/m2/day) > Value
[float] Mean solar input with single axis tracking in kWh/m2/day.

Hourly Irradiation > Mean solar input, two axis tracking (kWh/m2/day) > Value
[float] Mean solar input with two axis tracking in kWh/m2/day.

pyH2A.Plugins.Hourly_Irradiation_Plugin.**calculate_PV_power_ratio**(*file_name*, *module_tilt*, *array_azimuth*, *nominal_operating_temperature*, *temperature_coefficient*, *mismatch_derating*, *dirt_derating*)

Calculation based on Chang 2020, https://doi.org/10.1016/j.xcrp.2020.100209 SAT: horzontal single axis tracking DAT: dual axis tracking, no diffuse radiation

pyH2A.Plugins.Hourly_Irradiation_Plugin.**converter_function**(*string*)

Converter function for datetime of hourly irradiation data.

pyH2A.Plugins.Hourly_Irradiation_Plugin.**import_Chang_data**(*file_name*)

Import of Chang 2020 data, for debugging.

pyH2A.Plugins.Hourly_Irradiation_Plugin.**import_hourly_data**(*file_name*)

Imports hourly irradiation data and location coordinates from the *.csv* format provided by: https://re.jrc.ec.europa.eu/pvg_tools/en/#TMY. `@lru_cache` is used for fast repeated reads

4.6 Multiple_Modules_Plugin

Classes:

Multiple_Modules_Plugin(dcf, print_info)	Simulating mutliple plant modules which are operated together, assuming that only labor cost is reduced.

class pyH2A.Plugins.Multiple_Modules_Plugin.**Multiple_Modules_Plugin**(*dcf*, *print_info*)

Simulating mutliple plant modules which are operated together, assuming that only labor cost is reduced. Calculation of required labor to operate all modules, scaling down labor requirement to one module for subsequent calculations.

Parameters

Technical Operating Parameters and Specifications > Plant Modules > Value
[float or int] Number of plant modules considered in this calculation, `process_table()` is used.

Non-Depreciable Capital Costs > Solar Collection Area (m2) > Value
[float] Solar collection area for one plant module in m2, `process_table()` is used.

Fixed Operating Costs > area > Value
[float] Solar collection area in m2 that can be covered by one staffer.

Fixed Operating Costs > shifts > Value
[float or int] Number of 8-hour shifts (typically 3 for 24h operation).

Fixed Operating Costs > supervisor > Value
[float or int] Number of shift supervisors.

Returns

Fixed Operating Costs > staff > Value
[float] Number of 8-hour equivalent staff required for operating one plant module.

Methods:

required_staff(dcf)	Calculation of total required staff for all plant modules, then scaling down to staff requirements for one module.

required_staff(*dcf*)

Calculation of total required staff for all plant modules, then scaling down to staff requirements for one module.

4.7 PEC_Plugin

Classes:

PEC_Plugin(dcf, print_info)	Simulating H2 production using photoelectrochemical water splitting.

class pyH2A.Plugins.PEC_Plugin.**PEC_Plugin**(*dcf*, *print_info*)

Simulating H2 production using photoelectrochemical water splitting.

Parameters

Technical Operating Parameters and Specifications > Design Output per Day > Value
[float] Design output in (kg of H2)/day, `process_table()` is used.

PEC Cells > Cell Cost ($/m2) > Value
[float] Cost of PEC cells in $/m2.

PEC Cells > Lifetime (year) > Value
[float] Lifetime of PEC cells in years before replacement is required.

PEC Cells > Length (m) > Value
[float] Length of single PEC cell in m.

PEC Cells > Width (m) > Value
[float] Width of single PEC cell in m.

Land Area Requirement > Cell Angle (degree) > Value
[float] Angle of PEC cells from the ground, in degrees.

Land Area Requirement > South Spacing (m) > Value
[float] South spacing of PEC cells in m.

Land Area Requirement > East/West Spacing (m) > Value
[float] East/West Spacing of PEC cells in m.

Solar-to-Hydrogen Efficiency > STH (%) > Value
[float] Solar-to-hydrogen efficiency in percentage or as a value between 0 and 1.

Solar Input > Mean solar input (kWh/m2/day) > Value
[float] Mean solar input in kWh/m2/day, `process_table()` is used.

Solar Concentrator > Concentration Factor > Value
[float, optional] Concentration factor created by solar concentration module, which is used in combination with PEC cells. If "Solar Concentrator" is in dcf.inp, `process_table()` is used.

Returns

Non-Depreciable Capital Costs > Land required (acres) > Value
[float] Total land area required in acres.

Non-Depreciable Capital Costs > Solar Collection Area (m2) > Value
[float] Solar collection area in m2.

Planned Replacement > Planned Replacement PEC Cells > Cost ($)
[float] Total cost of replacing all PEC cells once.

Planned Replacement > Planned Replacement PEC Cells > Frequency (years)
[float] Replacement frequency of PEC cells in years, identical to PEC cell lifetime.

Direct Capital Costs - PEC Cells > PEC Cell Cost ($) > Value
[float] Total cost of all PEC cells.

PEC Cells > Number > Value
[float] Number of individual PEC cells required for design H2 output capacity.

Methods:

PEC_cost(dcf)	Calculation of cost per cell, number of required cells and total cell cost.
hydrogen_production(dcf)	Calculation of (kg of H2)/day produced by single PEC cell.
land_area(dcf)	Calculation of total required land area and solar collection area.

PEC_cost(*dcf*)

Calculation of cost per cell, number of required cells and total cell cost.

hydrogen_production(*dcf*)

Calculation of (kg of H2)/day produced by single PEC cell.

land_area(*dcf*)

Calculation of total required land area and solar collection area.

4.8 Photocatalytic_Plugin

Classes:

Photocatalytic_Plugin(dcf, print_info)	Simulating H2 production using photocatalytic water splitting in plastic baggie reactors.

class pyH2A.Plugins.Photocatalytic_Plugin.**Photocatalytic_Plugin**(*dcf*, *print_info*)

Simulating H2 production using photocatalytic water splitting in plastic baggie reactors.

Parameters

Technical Operating Parameters and Specifications > Design Output per Day > Value
[float] Design output in (kg of H2)/day, `process_table()` is used.

Reactor Baggies > Cost Material Top ($/m2) > Value
[float] Cost of baggie top material in $/m2.

Reactor Baggies > Cost Material Bottom ($/m2) > Value
[float] Cost of baggie bottom material in $/m2.

Reactor Baggies > Number of ports > Value
[int] Number of ports per baggie.

Reactor Baggies > Other Costs ($) > Value
[float] Other costs per baggie.

Reactor Baggies > Markup factor > Value
[float] Markup factor for baggies, typically > 1.

Reactor Baggies > Length (m) > Value
[float] Length of single baggie in m.

Reactor Baggies > Width (m) > Value
[float] Width of single baggie in m.

Reactor Baggies > Height (m) > Value
[float] Height of reactor baggie in m. In this simulation this value determines the height of the water level and hence is an important parameter ultimately determining the level of light absorption and total catalyst amount.

Reactor Baggies > Additional land area (%) > Value
[float] Additional land area required, percentage or value > 0. Calculated as: (1 + addtional land area) * baggie area.

Reactor Baggies > Lifetime (years) > Value
[float] Lifetime of reactor baggies in years before replacement is required.

Catalyst > Cost per kg ($) > Value
[float] Cost per kg of catalyst.

Catalyst > Concentration (g/L) > Value
[float] Concentration of catalyst in g/L.

Catalyst > Lifetime (years) > Value
[float] Lifetime of catalysts in year before replacement is required.

Catalyst > Molar Weight (g/mol) > Value
[float, optional] If the molar weight of the catalyst (in g/mol) is specified, homogeneous catalyst properties (TON, TOF etc. are calculated).

Catalyst > Molar Attenuation Coefficient (M^-1 cm^-1) > Value
[float, optional] If the molar attenuation coefficient (in M^-1 cm^-1) is specified (along with the molar weight), absorbance and the fraction of absorbed light are also calculated.

Solar-to-Hydrogen Efficiency > STH (%) > Value
[float] Solar-to-hydrogen efficiency in percentage or as a value between 0 and 1.

Solar Input > Mean solar input (kWh/m2/day) > Value
[float] Mean solar input in kWh/m2/day, `process_table()` is used.

Solar Input > Hourly (kWh/m2) > Value
[ndarray] Hourly irradiation data.

Returns

Non-Depreciable Capital Costs > Land required (acres) > Value
[float] Total land area required in acres.

Non-Depreciable Capital Costs > Solar Collection Area (m2) > Value
[float] Solar colelction area in m2.

Planned Replacement > Planned Replacement Catalyst > Cost ($)
[float] Total cost of completely replacing the catalyst once.

Planned Replacement > Planned Replacement Catalyst > Frequency (years)
[float] Replacement frequency of catalyst in years, identical to catalyst lifetime.

Planned Replacement > Planned Replacement Baggie > Cost ($)
[float] Total cost of replacing all baggies.

Planned Replacement > Planned Replacement Baggie > Frequency (years)
[float] Replacement frequency of baggies in year, identical to baggie lifetime.

Direct Capital Costs - Reactor Baggies > Baggie Cost ($) > Value
[float] Total baggie cost.

Direct Capital Costs - Photocatalyst > Catalyst Cost ($) > Value
[float] Total catalyst cost.

Reactor Baggies > Number > Value
[int] Number of individual baggies required for design H2 production capacity.

Catalyst > Properties > Value
[dict] Dictionary containing detailed catalyst properties calculated from provided parameters.

['Photocatalytic_Plugin'].catalyst_properties
[dict] Attribute containing catalyst properties dictionary.

Water Volume > Volume (liters) > Value
[float] Total water volume in liters.

Methods:

baggie_cost(dcf)	Calculation of cost per baggie, number of required baggies and total baggie cost.
catalyst_activity(dcf)	Calculation of detailed catalyst properties based on provided parameters.
catalyst_cost(dcf)	Calculation of individual baggie volume, catalyst amount per baggie, total catalyst amount and total catalyst cost.
hydrogen_production(dcf)	Calculation of hydrogen produced per day per baggie (in kg).
land_area(dcf)	Calculation of total required land area and solar collection area.

`baggie_cost`(*dcf*)

Calculation of cost per baggie, number of required baggies and total baggie cost.

`catalyst_activity`(*dcf*)

Calculation of detailed catalyst properties based on provided parameters. If "Molar Weight (g/mol)" is specified in "Catalyst" table properties of a homogeneous catalyst are also calculated. Furthermore, if "Molar Attenuation Coefficient (M^-1 cm^-1)" is also provided, the light absorption properties are calculated.

catalyst_cost(*dcf*)

Calculation of individual baggie volume, catalyst amount per baggie, total catalyst amount and total catalyst cost.

hydrogen_production(*dcf*)

Calculation of hydrogen produced per day per baggie (in kg).

land_area(*dcf*)

Calculation of total required land area and solar collection area.

4.9 Photovoltaic_Plugin

Classes:

Photovoltaic_Plugin(dcf, print_info)	Simulation of hydrogen production using PV + electrolysis.

class pyH2A.Plugins.Photovoltaic_Plugin.**Photovoltaic_Plugin**(*dcf*, *print_info*)

Simulation of hydrogen production using PV + electrolysis.

Parameters

Financial Input Values > construction time > Value
[int] Construction time of hydrogen production plant in years.

Irradiation Used > Data > Value
[str or ndarray] Hourly power ratio data for electricity production calculation. Either a path to a text file containing the data or ndarray. A suitable array can be retrieved from "Hourly Irradiation > *type of tracking* > Value".

CAPEX Multiplier > Multiplier > Value
[float] Multiplier to describe cost reduction of PV and electrolysis CAPEX for every ten-fold increase of power relative to CAPEX reference power. Based on the multiplier the CAPEX scaling factor is calculated as: multiplier ^ (number of ten-fold increases). A value of 1 leads to no CAPEX reduction, a value < 1 enables cost reduction.

Electrolyzer > Nominal Power (kW) > Value
[float] Nominal power of electrolyzer in kW.

Electrolyzer > CAPEX Reference Power (kW) > Value
[float] Reference power of electrolyzer in kW for cost reduction calculation.

Electrolyzer > Power requirement increase per year > Value
[float] Electrolyzer power requirement increase per year due to stack degradation. Percentage or value > 0. Increase calculated as: (1 + increase per year) ^ year.

Electrolyzer > Minimum capacity > Value
[float] Minimum capacity required for electrolyzer operation. Percentage or value between 0 and 1.

Electrolyzer > Conversion efficiency (kg H2/kWh) > Value
[float] Electrical conversion efficiency of electrolyzer in (kg H2)/kWh.

Electrolyzer > Replacement time (h) > Value
[float] Operating time in hours before stack replacement of electrolyzer is required.

Photovoltaic > Nominal Power (kW) > Value
[float] Nominal power of PV array in kW.

Photovoltaic > CAPEX Reference Power (kW) > Value
[float] Reference power of PV array for cost reduction calculations.

Photovoltaic > Power loss per year > Value
[float] Reduction in power produced by PV array per year due to degradation. Percentage or value > 0. Reduction calculated as: (1 - loss per year) ^ year.

Photovoltaic > Efficiency > Value
[float] Power conversion efficiency of used solar cells. Percentage or value between 0 and 1.

Returns

Technical Operating Parameters and Specifications > Plant Design Capacity (kg of H2/day) > Value
[float] Plant design capacity in (kg of H2)/day calculated from installed PV + electrolysis power capacity and hourly irradiation data.

Technical Operating Parameters and Specifications > Operating Capacity Factor (%) > Value
[float] Operating capacity factor is set to 1 (100%).

Planned Replacement > Electrolyzer Stack Replacement > Frequency (years)
[float] Frequency of electrolyzer stack replacements in years, calculated from replacement time and hourly irradiation data.

Electrolyzer > Scaling Factor > Value
[float] CAPEX scaling factor for electrolyzer calculated based on CAPEX multiplier, reference and nominal power.

Photovoltaic > Scaling Factor > Value
[float] CAPEX scaling factor for PV array calculated based on CAPEX multiplier, reference and nominal power.

Non-Depreciable Capital Costs > Land required (acres) > Value
[float] Total land required in acres.

Non-Depreciable Capital Costs > Solar Collection Area (m2) > Value
[float] Solar collection area in m2.

Methods:

calculate_H2_production(dcf)	Using hourly irradiation data and electrolyzer as well as PV array parameters, H2 production is calculated.
calculate_area(dcf)	Area requirement calculation assuming 1000 W/m2 peak power.
calculate_electrolyzer_power_demand(dcf, year)	Calculation of yearly increase in electrolyzer power demand.
calculate_photovoltaic_loss_correction(dcf, ...)	Calculation of yearly reduction in electricity production by PV array.
calculate_scaling_factors(dcf)	Calculation of electrolyzer and PV CAPEX scaling factors.
calculate_stack_replacement(dcf)	Calculation of stack replacement frequency for electrolyzer.
scaling_factor(dcf, power, reference)	Calculation of CAPEX scaling factor based on nominal and reference power.

calculate_H2_production(*dcf*)

Using hourly irradiation data and electrolyzer as well as PV array parameters, H2 production is calculated.

calculate_area(*dcf*)

Area requirement calculation assuming 1000 W/m2 peak power.

calculate_electrolyzer_power_demand(*dcf*, *year*)

Calculation of yearly increase in electrolyzer power demand.

calculate_photovoltaic_loss_correction(*dcf*, *data*, *year*)

Calculation of yearly reduction in electricity production by PV array.

calculate_scaling_factors(*dcf*)

Calculation of electrolyzer and PV CAPEX scaling factors.

calculate_stack_replacement(*dcf*)

Calculation of stack replacement frequency for electrolyzer.

scaling_factor(*dcf*, *power*, *reference*)

Calculation of CAPEX scaling factor based on nominal and reference power.

4.10 Production_Scaling_Plugin

Classes:

Production_Scaling_Plugin(dcf, print_info)	Calculation of plant output and potential scaling.

class pyH2A.Plugins.Production_Scaling_Plugin.**Production_Scaling_Plugin**(*dcf*, *print_info*)

Calculation of plant output and potential scaling.

Parameters

Technical Operating Parameters and Specifications > Plant Design Capacity (kg of H2/day) > Value
[float] Plant design capacity in kg of H2/day, `process_table()` is used.

Technical Operating Parameters and Specifications > Operating Capacity Factor (%) > Value
[float] Operating capacity factor in %, `process_table()` is used.

Technical Operating Parameters and Specifications > Maximum Output at Gate > Value
[float, optional] Maximum output at gate in (kg of H2)/day, `process_table()` is used. If this parameter is not specified it defaults to *Plant Design Capacity (kg of H2/day)*.

Technical Operating Parameters and Specifications > New Plant Design Capacity (kg of H2/day) > Value
[float, optional] New plant design capacity in kg of H2/day to calculate scaling, which overwrites possible Scaling Ratio, `process_table()` is used.

Technical Operating Parameters and Specifications > Scaling Ratio > Value
[float, optional] Scaling ratio which is multiplied by current plant design capacity to obtain scaled plant size, `process_table` is used.

Technical Operating Parameters and Specifications > Capital Scaling Exponent > Value
[float, optional] Exponent to calculate capital scaling factor, `process_table()` is used. Defaults to 0.78.

Technical Operating Parameters and Specifications > Labor Scaling Exponent > Value
[float, optional] Exponent to calculcate labor scaling factor, `process_table()` is used. Defaults to 0.25.

Returns

Technical Operating Parameters and Specifications > Design Output per Day > Value
[float] Design output in (kg of H2)/day.

Technical Operating Parameters and Specifications > Max Gate Output per Day > Value
[float] Maximum gate ouput in (kg of H2)/day.

Technical Operating Parameters and Specifications > Output per Year > Value
[float] Yearly output taking operating capacity factor into account, in (kg of H2)/year.

Technical Operating Parameters and Specifications > Output per Year at Gate > Value
[float] Actual yearly output at gate, in (kg of H2)/year.

Technical Operating Parameters and Specifications > Scaling Ratio > Value
[float or None] Returned if New Plant Design Capacity was specified.

Scaling > Capital Scaling Factor > Value
[float or None] Returned if scaling is active (*Scaling Ratio* or *New Plant Design Capacity (kg of H2/day)* specified).

Scaling > Labor Scaling Factor > Value
[float or None] Returned if scaling is active (*Scaling Ratio* or *New Plant Design Capacity (kg of H2/day)* specified).

Notes

To scale capital or labor costs, a path to *Scaling > Capital Scaling Factor > Value* or *Scaling > Labor Scaling Factor > Value* has to specified for the respective table entry.

Methods:

calculate_output(dcf)	Calculation of yearly output in kg and yearly output at gate in kg.
calculate_scaling(dcf, print_info)	Calculation of scaling if scaling is requested (either *New Plant Design Capacity (kg of H2/day)* or *Scaling Ratio* was provided).

calculate_output(*dcf*)

Calculation of yearly output in kg and yearly output at gate in kg.

calculate_scaling(*dcf*, *print_info*)

Calculation of scaling if scaling is requested (either *New Plant Design Capacity (kg of H2/day)* or *Scaling Ratio* was provided). Otherwise returns regular design output and output at gate per day in (kg H2).

4.11 Replacement_Plugin

Classes:

Planned_Replacement(dictionary, key, dcf)	Individual planned replacement objects.
Replacement_Plugin(dcf, print_info)	Calculating yearly overall replacement costs based on one-time replacement costs and frequency.

class pyH2A.Plugins.Replacement_Plugin.**Planned_Replacement**(*dictionary*, *key*, *dcf*)

Individual planned replacement objects.

Methods

calculate_yearly_cost:	Calculation of yearly costs from one-time cost and replacement frequency.

Methods:

calculate_yearly_cost(dictionary, key, dcf)	Calculation of yearly replacement costs.

calculate_yearly_cost(*dictionary*, *key*, *dcf*)

Calculation of yearly replacement costs.

Replacement costs are billed annually, replacements which are performed at a non-integer rate are corrected using non_integer_correction.

class pyH2A.Plugins.Replacement_Plugin.**Replacement_Plugin**(*dcf*, *print_info*)

Calculating yearly overall replacement costs based on one-time replacement costs and frequency.

Parameters

Planned Replacement > [...] > Frequency (years)
[float] Replacement frequency of [...] in years. Iteration over all entries in *Planned Replacement* table. No path key available.

Planned Replacement > [...] > Cost ($)
[float] One-time replacement cost of [...]. Iteration over all entries in *Planned Replacement* table. Path key is 'Path'.

[...] Unplanned Replacement [...] >> Value
[float] `sum_all_tables()` is used.

Returns

[...] Unplanned Replacement [...] > Summed Total > Value
[float] Summed total for each individual table in "Unplanned Replacement" group.

Replacement > Total > Value
[ndarray] Total inflated replacement costs (sum of *Planned Replacement* entries and unplanned replacement costs).

Methods:

calculate_planned_replacement(dcf)	Calculation of yearly replacement costs by iterating over all entries of *Planned Replacement*.
initialize_yearly_costs(dcf)	Initializes ndarray filled with zeros with same length as dcf.inflation_factor.
unplanned_replacement(dcf, print_info)	Calculating unplanned replacement costs by appling `sum_all_tables()` to "Unplanned Replacement" group.

calculate_planned_replacement(*dcf*)

Calculation of yearly replacement costs by iterating over all entries of *Planned Replacement*.

initialize_yearly_costs(*dcf*)

Initializes ndarray filled with zeros with same length as dcf.inflation_factor.

unplanned_replacement(*dcf*, *print_info*)

Calculating unplanned replacement costs by appling `sum_all_tables()` to "Unplanned Replacement" group.

4.12 Solar_Concentrator_Plugin

Classes:

Solar_Concentrator_Plugin(dcf, print_info)	Simulation of solar concentration (used in combination with PEC cells).

class pyH2A.Plugins.Solar_Concentrator_Plugin.**Solar_Concentrator_Plugin**(*dcf*, *print_info*)

Simulation of solar concentration (used in combination with PEC cells).

Parameters

Solar Concentrator > Concentration Factor > Value
[float] Concentration factor of solar concentration, value > 1.

Solar Concentrator > Cost ($/m2) > Value
[float] Cost of solar concentrator in $/m2.

PEC Cells > Number > Value
[float] Number of PEC cells required for design H2 production capacity.

Land Area Requirement > South Spacing (m) > Value
[float] South spacing of solar concentrators in m.

Land Area Requirement > East/West Spacing (m) > Value
[float] East/West Spacing (m) of solar concentrators in m.

Non-Depreciable Capital Costs > Solar Collection Area (m2) > Value
[float] Total solar collection area in m2.

Returns

Non-Depreciable Capital Costs > Land required (m2) > Value
[float] Total land requirement in m2.

Non-Depreciable Capital Costs > Land required (acres) > Value
[float] Total land requirement in acres.

Non-Depreciable Capital Costs > Solar Collection Area (m2) > Value
[float] Total solar collection area in m2.

Direct Capital Costs - Solar Concentrator > Solar Concentrator Cost ($) > Value
[float] Total cost of all solar concentrators.

Methods:

calculate_cost(dcf)	Calculation of solar concentrator cost based on cost per m2 and total solar collection area.
land_area(dcf)	Calculation of solar collection area by multiplying concentration factor by supplied (unconcentrated) solar collection area.

calculate_cost(*dcf*)

Calculation of solar concentrator cost based on cost per m2 and total solar collection area.

land_area(*dcf*)

Calculation of solar collection area by multiplying concentration factor by supplied (unconcentrated) solar collection area. Calculation of total land area requirement based on number of PEC cells and spacing of solar concentrators.

4.13 Solar_Thermal_Plugin

Classes:

Solar_Thermal_Plugin(dcf, print_info)	Simulation of hydrogen production using solar thermal water splitting.

class pyH2A.Plugins.Solar_Thermal_Plugin.**Solar_Thermal_Plugin**(*dcf*, *print_info*)

Simulation of hydrogen production using solar thermal water splitting.

Parameters

Technical Operating Parameters and Specifications > Design Output per Day > Value
[float] Design output of hydrogen production plant per day in kg.

Solar-to-Hydrogen Efficiency > STH (%) > Value
[float] Solar-to-Hydrogen Efficiency of thermal water splitting process. Percentage of value between 0 and 1.

Solar Input > Mean solar input (kWh/m2/day) > Value
[float] Mean solar input in kWh/m2/day.

Non-Depreciable Capital Costs > Additional Land Area (%) > Value
[float] Additional land area required. Percentage or value > 0. Calculated as: (1 + Addtional Land Area) * solar collection area.

Returns

Non-Depreciable Capital Costs > Land required (acres) > Value
[float] Total land requirement in acres.

Methods:

calculate_land_area(dcf)	Calculation of required land area based on solar input, solar-to-hydrogen efficiency and addtional land are requirements.

calculate_land_area(*dcf*)

Calculation of required land area based on solar input, solar-to-hydrogen efficiency and addtional land are requirements.

4.14 Variable_Operating_Cost_Plugin

Classes:

Utility(dictionary, dcf)	Individual utility objects.
Variable_Operating_Cost_Plugin(dcf, print_info)	Calculation of variable operating costs.

class pyH2A.Plugins.Variable_Operating_Cost_Plugin.**Utility**(*dictionary*, *dcf*)

Individual utility objects.

Methods

calculate_cost_per_kg_H2:	Calculation of utility cost per kg of H2 with inflation correction.

Methods:

calculate_cost_per_kg_H2(dictionary, dcf)	Calculation of utility cost per kg of H2 with inflation correction.

calculate_cost_per_kg_H2(*dictionary*, *dcf*)

Calculation of utility cost per kg of H2 with inflation correction.

class pyH2A.Plugins.Variable_Operating_Cost_Plugin.**Variable_Operating_Cost_Plugin**(*dcf*, *print_info*)

Calculation of variable operating costs.

Parameters

Technical Operating Parameters and Specifications > Output per Year > Value
[float] Yearly output taking operating capacity factor into account, in (kg of H2)/year.

Utilities > [...] > Cost
[float, ndarray or str] Cost of utility (e.g. $/kWh for electricity). May be either a float, a ndarray with the same length as *dcf.inflation_correction* or a textfile containing cost values (cost values have to be in second column).

Utilities > [...] > Usage per kg H2
[float] Usage of utility per kg H2 (e.g. kWh/(kg of H2) for electricity).

Utilities > [...] > Price Conversion Factor
[float] Conversion factor between cost and usage units. Should be set to 1 if no conversion is required.

Utilities > [...] > Path
[str, optional] Path for *Cost* entry.

Utilities > [...] > Usage Path
[str, optional] Path for *Usage per kg H2* entry.

[...] Other Variable Operating Operating Cost [...] >> Value
[float] `sum_all_tables()` is used.

Returns

[...] Other Variable Operating Cost [...] > Summed Total > Value
[float] Summed total for each individual table in "Other Variable Operating Cost" group.

Variable Operating Costs > Total > Value
[ndarray] Sum of inflation corrected utilities costs and other variable operating costs.

Variable Operating Costs > Utilities > Value
[ndarray] Sum of inflation corrected utilities costs.

Variable Operating Costs > Other > Value
[float] Sum of *Other Variable Operating Cost* entries.

Methods:

calculate_utilities_cost(dcf)	Iterating over all utilities and computing summed yearly costs.
other_variable_costs(dcf, print_info)	Applying `sum_all_tables()` to "Other Variable Operating Cost" group.

`calculate_utilities_cost`(*dcf*)

Iterating over all utilities and computing summed yearly costs.

`other_variable_costs`(*dcf*, *print_info*)

Applying `sum_all_tables()` to "Other Variable Operating Cost" group.

CHAPTER

FIVE

ANALYSIS

5.1 Cost_Contributions_Analysis

Classes:

Cost_Contributions_Analysis(input_file)	Reading cost contributions from DCF output and plotting it.

class pyH2A.Analysis.Cost_Contributions_Analysis.**Cost_Contributions_Analysis**(*input_file*)

Reading cost contributions from DCF output and plotting it.

Notes

Cost_Contributions_Analysis does not require any input, it is sufficient to include "# Cost_Contributions_Analysis" in input file. Contributions for plotting have to provided as a dictionary with three components: *Data* containing the name and value for each contribution. *Total* containing the total value *Table Group*, containing the name of overarching group of values, which are shown. Dictionaries with this structure can be automatically generated by `sum_all_tables` function. If the name of a contribution contains a '-', the string will be split and only the part after the '-' will be displayed.

Methods:

cost_breakdown_plot([ax, figure_lean, ...])	Plotting cost breakdown plot.

cost_breakdown_plot(*ax=None*, *figure_lean=True*, *plugin=None*, *plugin_property=None*, *label_offset=5.5*, *x_label_string='Cost / USD'*, *x_label_string_H2='Levelized cost / USD per kg H_{2}'*, *plot_kwargs={}*, ***kwargs*)

Plotting cost breakdown plot.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

plugin
[str or None, optional] Selection of plugin from which contributions data is to be read. If None, defaults to overall H2 cost contributions.

plugin_property: str or None, optional
Specific property of plugin from which contributions data is to be read.

label_offset
[float, optional] Offset for contributions labels.

x_label_string
[str, optional] String for x axis label.

x_label_string_H2
[str, optional] String for x axis label when overall H2 cost breakdown is plotted

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

Notes

Names of contributions are split at '-' symbols and only the last part is displayed. For example: *Direct Capital Costs - Installation* is displayed only as *Installation*.

5.2 Sensitivity_Analysis

Classes:

Sensitivity_Analysis(input_file)	Sensitivity analysis for multiple parameters.

class pyH2A.Analysis.Sensitivity_Analysis.**Sensitivity_Analysis**(*input_file*)

Sensitivity analysis for multiple parameters.

Parameters

Sensitivity_Analysis > [...] > Name
[str] Display name for parameter, e.g. used for plot labels.

Sensitivity_Analysis > [...] > Type
[str] Type of parameter values. If *Type* is 'value', provided values are used as is. If *Type* is 'factor', provided values are multiplied with base value of parameter in input file.

Sensitivity_Analysis > [...] > Values
[str] Value pair to be used for sensitivity analysis. One value should be higher than the base value, the other should be lower. Specified in following format: value A; value B (order is irrelevant). E.g. '0.3; 10'.

Notes

Sensitivity_Analysis contains parameters which are to be varied in sensitivity analysis. First column specifies path to parameter in input file (top key > middle key > bottom key format, e.g. Catalyst > Cost per kg ($) > Value). Order of parameters is not relevant.

Methods:

perform_sensitivity_analysis([format_cutoff])	Perform sensitivity analysis.
sensitivity_box_plot([ax, figure_lean, ...])	Plot sensitivity box plot.
sort_h2_cost_values(data)	Sort H2 cost values.

perform_sensitivity_analysis(*format_cutoff=7*)

Perform sensitivity analysis.

Parameters

format_cutoff
[int] Length of number string above which dyanmic formatting is applied.

sensitivity_box_plot(*ax=None*, *figure_lean=True*, *height=0.8*, *label_offset=0.1*, *lim_extra=0.2*, *format_cutoff=7*, *plot_kwargs={}*, ***kwargs*)

Plot sensitivity box plot.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

height
[float, optional] Height of bars.

label_offset
[float, optional] Offset for bar labels.

lim_extra
[float, optional] Fractional increase of x axis limits.

format_cutoff
[int] Length of number string above which dyanmic formatting is applied.

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

Notes

In plot, parameters are sorted by descending cost increase magnitude.

sort_h2_cost_values(*data*)

Sort H2 cost values.

5.3 Waterfall_Analysis

Classes:

Waterfall_Analysis(input_file)	Perform waterfall analysis to study the compounded effect of changing multiple parameters.

class pyH2A.Analysis.Waterfall_Analysis.**Waterfall_Analysis**(*input_file*)

Perform waterfall analysis to study the compounded effect of changing multiple parameters.

Parameters

Waterfall_Analysis > [...] > Name
[str] Display name for parameter, e.g. used for plot labels.

Waterfall_Analysis > [...] > Type
[str] Type of parameter values. If *Type* is 'value', provided values are used as is. If *Type* is 'factor', provided values are multiplied with base value of parameter in input file.

Waterfall_Analysis > [...] > Value
[float] New value or factor for parameter.

Waterfall_Analysis > [...] > Show Percent
[bool or str, optional] If there is any entry for *Show Percent* the parameter will be displayed as a percentage value in waterfall chart.

Notes

Waterfall_Analysis contains parameters which are to be varied in waterfall analysis. First column specifies path to parameter in input file (top key > middle key > bottom key format, e.g. Catalyst > Cost per kg ($) > Value). Order of varied parameters determines in which order they are applied. In the order they are provided, each parameter is changed to the provided value. The relative change of introducing each change is computed, and the new H2 cost (compound result of applying all changes) is calculated.

Methods:

modify_inp_run_dcf(inp, dic, output)	Modification of *inp* with values from *dic* and running discounted cash flow analysis.
perform_waterfall_analysis()	Perform waterfall analysis
plot_waterfall_chart([ax, figure_lean, ...])	Plot waterfall chart.
show_percent(value, dic)	Displaying provided *value* as percentage if *Show Percent* is in dictionary.

modify_inp_run_dcf(*inp*, *dic*, *output*)

Modification of *inp* with values from *dic* and running discounted cash flow analysis.

`perform_waterfall_analysis()`

Perform waterfall analysis

`plot_waterfall_chart(ax=None, figure_lean=True, width=0.7, connection_width=1.0, label_offset=20, plot_sorted=False, plot_kwargs={}, **kwargs)`

Plot waterfall chart.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

width
[float, optional] Width of bars in waterfall chart.

connection_width
[float, optional] Width of lines connecting bars.

label_offset
[float, optional] Offset of label for bars.

plot_sorted
[bool, optional] If *plot_sorted* is True, bars are plotted in a sorted manner. If it False, they are plotted in the order they are provided in the input file.

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

`show_percent(value, dic)`

Displaying provided *value* as percentage if *Show Percent* is in dictionary.

5.4 Monte_Carlo_Analysis

Classes:

Monte_Carlo_Analysis(input_file)	Monte Carlo analysis of a techno-economic model.

Functions:

calculate_distance(data, parameters, selection)	Distance of datapoints to reference is calculated using the specified metric.
coordinate_position(x_reference, x_values, ...)	Determine correct position for base case label in *plot_colored_scatter()*.
divide_into_batches(array, batch_size)	Divide provided array into batches of size batch_size for parallel processing
extend_limits(limits_original, extension)	Extend limits_original in both directions by muliplying with extensions
normalize_parameter(parameter, base, limit)	Linear of log normalization of parameter (float or array) based on base and limit values.
select_non_reference_value(reference, values)	Select value from values which is not the reference one.

class pyH2A.Analysis.Monte_Carlo_Analysis.**Monte_Carlo_Analysis**(*input_file*)

Monte Carlo analysis of a techno-economic model.

Parameters

Monte_Carlo_Analysis > Samples > Value
[int] Number of samples for Monte Carlo analysis.

Monte_Carlo_Analysis > Target Price Range ($) > Value
[str] Target price range for H2 specfied in the following format: lower value; higher value (e.g. "1.5: 1.54").

Monte_Carlo_Analysis > Output File > Value
[str, optional] Path to location where output file containing Monte Carlo analysis results should be saved.

Monte_Carlo_Analysis > Input File > Value
[str, optional] Path to location of file containing Monte Carlo analysis results that should be read.

Parameters - Monte_Carlo_Analysis > [...] > Name
[str] Display name for parameter, e.g. used for axis labels.

Parameters - Monte_Carlo_Analysis > [...] > Type
[str] Type of parameter values. If *Type* is 'value', provided values are used as is. If *Type* is 'factor', provided values are multiplied with base value of parameter in input file.

Parameters - Monte_Carlo_Analysis > [...] > Values
[str] Value range for parameter in Monte Carlo analysis. Specified in following format: upper limit; lower limit (order is irrelevant). Instead of actual values, 'Base' or 'Reference' can be used to retrieve base value of parameter in input file as one of the values. E.g. 'Base; 20%'.

Parameters - Monte_Carlo_Analysis > [...] > File Index
[int, optional] If Monte Carlo results are read from a provided input file, *File Index* for each parameter can be specified. *File Index* refers to the column position of each parameter in the read input file. This mapping allows for changing the display name and position of a parameter in the *Parameters - Monte_Carlo_Analysis* analysis table and still ensure correct mapping of the read results.

Notes

Parameters - Monte_Carlo_Analysis contains parameters which are to be varied in Monte Carlo Analysis. First column specifies path to parameter in input file (top key > middle key > bottom key format, e.g. Catalyst > Cost per kg ($) > Value). Order of parameters can be changed, which for example affects the mapping onto different axis in *plot_colored_scatter* (first parameter is on x axis, second on y axis, etc.).

Methods:

check_parameter_integrity(values)	Checking that parameters in *self.results* are within ranges specified in *self.parameters['Values']*.
determine_principal_components()	Converting parameters to list sorted by input index.
development_distance([metric, ...])	Calculation of development distance for models within target price range.
full_distance_cost_relationship([metric, ...])	Calculation of development distance for all datapoints from Monte Carlo Analysis and calculation of Savitzky-Golay filter.
generate_parameter_string_table([...])	String of parameter table is generated, used in *self.render_parameter_table*.
perform_full_monte_carlo()	Monte Carlo analysis is performed based on random parameter variations in *self.values*.
perform_h2_cost_calculation(values)	H2 cost calculation for provided parameter values is performed.
perform_monte_carlo_multiprocessing(values)	Monte Carlo analysis is performed with multiprocessing parallelization across all available CPUs.
plot_colored_scatter([limit_extension, ...])	Plotting colored scatter plot showing all models within target price range.
plot_colored_scatter_3D([limit_extension, ...])	3D colored scatter plot of models within target price range.
plot_complete_histogram([bins, xlim_low, ...])	Complete histogram of price distribution from Monte Carlo analysis
plot_distance_cost_relationship([ax, ylim, ...])	Plotting relationship of development distance and H2 cost.
plot_distance_histogram([ax, bins, ...])	Plotting development distances as histogram.
plot_target_parameters_by_distance([ax, ...])	
process_parameters()	Monte Carlo Analysis parameters are read from 'Monte Carlo Analysis - Parameters' in *self.inp* and processed.
read_results(file_name)	Reads Monte Carlo simulation results from *file_name*.
render_parameter_table(ax[, xpos, ypos, ...])	Rendering table of parameters which are varied during Monte Carlo analysis.
save_results(file_name)	Results of Monte Carlo simulation are saved in *file_name* and a formatted header is added.
target_price_2D_region([grid_points])	Determining largest region spanned by first two parameters within which target prices can be achieved.
target_price_components()	Monte Carlo simulation results are sorted by H2 cost and the entries of *self.results* with a H2 cost within the specified target price range are stored in *self.target_price_data*.

check_parameter_integrity(*values*)

Checking that parameters in *self.results* are within ranges specified in *self.parameters['Values']*.

`determine_principal_components()`

Converting parameters to list sorted by input index.

`development_distance(`*metric='cityblock'*, *log_normalize=False*, *sum_distance=False*`)`

Calculation of development distance for models within target price range.

Parameters

metric: str, optional
Metric used for distance calculation, defaults to *cityblock*.

Returns

self.distances: ndarrray
Array containing distances for models within target price range.

Notes

The euclidean or cityblock distance in n-dimensional space of each Monte Carlo simulation datapoint within the target price range to the reference point is calculated and stored in self.distances. Parameter ranges and the reference parameters are scaled to be within a n-dimensional unit cube. Distances are normalized by the number of dimensions, so that the maximum distance is always 1.

`full_distance_cost_relationship(`*metric='cityblock'*, *reduction_factor=25*, *poly_order=4*, *log_normalize=False*, *sum_distance=False*`)`

Calculation of development distance for all datapoints from Monte Carlo Analysis and calculation of Savitzky-Golay filter.

Parameters

metric
[str, optional] Distance metric used for `calculate_distance()`

reduction_factor
[int, optional] Determines window size for Savitzky-Golay filter.

poly_order
[int, optional] Order of polynomial for Savitzky-Golay filter

Returns

self.results_distances_sorted
[ndarray] Sorted array of distances for all datapoints from Monte Carlo Analysis.

self.distances_cost_savgol
[ndarray] Savitzky-Golay filter results.

`generate_parameter_string_table(`*base_string='Base'*, *limit_string='Limit'*, *format_cutoff=6*`)`

String of parameter table is generated, used in *self.render_parameter_table*.

Parameters

base_string
[str, optional] String used to label base column.

limit_string
[str, optinal] String used to label limit column.

format_cutoff
[int] Length of number string at which it is converted to scientific/millified representation.

`perform_full_monte_carlo()`

Monte Carlo analysis is performed based on random parameter variations in *self.values*.

`perform_h2_cost_calculation(values)`

H2 cost calculation for provided parameter values is performed.

Parameters

values
[ndarray] Array containing parameter variations.

Returns

h2_cost
[ndarray] 1D array of H2 cost values for each set of parameters.

Notes

Performs H2 cost calulation by modifying a copy of self.inp based on the provided values and *self.parameters*. The modified copy of *self.inp* is then passed to *Discounted_Cash_Flow()*. A parameter value can be either a value replacing the existing one in self.inp (Type = value) or it can be a factor hich will be multiplied by the existing value.

`perform_monte_carlo_multiprocessing(values, return_full_array=True)`

Monte Carlo analysis is performed with multiprocessing parallelization across all available CPUs.

Parameters

values
[ndarray] 2D array containing parameters variations which are to be evaluated.

return_full_array
[bool, optional] If *return_full_array* is True, the full 2D array containing parameter variations and H2 cost is returned. Otherwise, a 1D array containing only H2 cost values is returned.

Returns

full_array
[ndarray] 2D array containing parameter variations and H2 cost values.

h2_cost
[ndarray] 1D array containing H2 costvalues.

`plot_colored_scatter(limit_extension=0.03, title_string='Target cost range: ', base_string='Base', image_kwargs={}, plot_kwargs={}, **kwargs)`

Plotting colored scatter plot showing all models within target price range.

Parameters

limit_extension: float, optional
Amount of limit extension of axes as fraction of axis range.

title_string
[str, optional] String for title.

base_string
[str, optional] String to label base case datapoint.

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned.

Notes

x, y and color axis are determined by *determine_principal_components()*, with pc[0] being the x axis, pc[1] the y axis and pc[2] the color axis. Order can be changed by changing order of parameters in input file.

plot_colored_scatter_3D(*limit_extension=0.03*, *title_string='Target cost range: '*, ***kwargs*)

3D colored scatter plot of models within target price range.

Parameters

limit_extension: float, optional
Amount of limit extension of axes as fraction of axis range.

title_string: str, optional
Title string.

Returns

fig: matplotlib.figure
Figure object.

plot_complete_histogram(*bins=None*, *xlim_low=None*, *xlim_high=None*, *xlabel_string='Levelized H_{2} Cost / \\$/kg'*, *ylabel_string='Normalized Frequency'*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

Complete histogram of price distribution from Monte Carlo analysis

Parameters

bins
[int, optional] Number of bins for histogram. If *None*, bins is calculated from the size of *self.results*.

xlim_low
[float or None, optional] Lower x axis limit.

xlim_high
[float or None, optional] Higher x axis limit.

xlabel_string
[str, optional] String for x axis label.

ylabel_string
[str, optional] String for y axis label.

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned.

`plot_distance_cost_relationship`(*ax=None*, *ylim=None*, *xlim=None*, *figure_lean=True*, *parameter_table=True*, *legend_loc='upper left'*, *log_scale=False*, *xlabel_string='Development distance'*, *ylabel_string='Levelized H_{2} cost / \\$/kg'*, *linewidth=1.5*, *markersize=0.2*, *marker_alpha=0.2*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

Plotting relationship of development distance and H2 cost.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

ylim
[array, optional] Ordered limit values for y axis. Default is None.

xlim
[array, optional] Ordered limit values for x axis. Default is None.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

parameter_table
[bool, optional] If parameter_table is True, the parameter table is shown in the plot.

legend_loc
[str, optional] Controls location of legend in plot. Defaults to 'upper left'.

log_scale
[bool, optional] If log_scale is True, the y axis will use a log scale.

xlabel_string
[str, optional] String for x axis label.

ylabel_string
[str, optional] String for y axis label.

linewidth
[float, optional] Line width for smoothed trendline.

markersize
[float, optional] Size of markers in scatter plot.

marker_alpha
[float, optional] Transparency of markers in scatter plot (0: maximum transpareny, 1: no transparency).

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

`plot_distance_histogram`(*ax=None*, *bins=25*, *figure_lean=True*, *xlabel=False*, *title=True*, *xlabel_string='Development distance'*, *ylabel_string='Frequency'*, *title_string='Target cost range:'*, *show_parameter_table=True*, *show_mu=True*, *mu_x=0.2*, *mu_y=0.5*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

Plotting development distances as histogram.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

bins
[int, optional] Number of bins for histogram.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

xlabel
[bool, optional] Flag to control if x axis label is displayed or not.

title
[bool, optional] Flag to control if title is displayed or not.

title_string
[str, optional] String for title.

xlabel_string
[str, optional] String for x axis label.

ylabel_string
[str, optional] String for y axis label.

show_parameter_table
[bool, optional] Flag to control if parameter table is shown.

show_mu
[bool, optional] Flag to control if mu and sigma values of normal distribution are shown.

mu_x
[float, optional] x axis coordinate of shown mu and sigma values in axis coordinates.

mu_y
[float, optional] y axis coordinate of shown mu and sigma values in axis coordinates.

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

`plot_target_parameters_by_distance`(*ax=None*, *figure_lean=True*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

`process_parameters`()

Monte Carlo Analysis parameters are read from 'Monte Carlo Analysis - Parameters' in *self.inp* and processed.

Notes

The ranges for each parameter are defined in *Values* column as ';' seperated entries. Entries can either be a number, a path, or a *special_value* such as *Base* or *Reference*. If such a *special_value* is specified, the base value of that parameter is retrieved from *self.inp*. Parameter information is stored in *self.parameters* attribute. Based on the ranges for each parameter, random values (uniform distribution) are generated and stored in the *self.values* attribute. The target price range is read from *self.inp* file and stored in *self.target_price_range* attribute.

`read_results`(*file_name*)

Reads Monte Carlo simulation results from *file_name*.

Parameters

file_name
[str] Path to file containing Monte Carlo simulation results.

Returns

self.results
[ndarray] Array containing parameters and H2 cost for each model.

self.parameters
[dict] Dictionary containing information on varied parameters.

self.target_price_range
[ndarray] Selected target price range from *self.inp*.

Notes

Assumes formatting created by *self.save_results()`function. Header must contain name of parameters, path to parameters in input file, type of parameter and value range. The header is processed to retrieve these atrribtues and stores them in `self.parameters*. Reference values for each parameter and target price range are read from self.inp The order of parameters is also read from *self.inp* and stored in *self.parameters* as *Input Index*. If the name of a parameter has been changed in *self.inp* and is different from the parameter stored in *file_name*, it is checked whether a *File Index* is specified, which allows for mapping of renamed parameter to parameter stored in *File Index*. If *File Index* is specified, the existing parameter at this position in *File Index* is renamed to the specified name.

render_parameter_table(*ax*, *xpos=1.05*, *ypos=0.0*, *height=1.0*, *colWidths=[0.55, 0.25, 0.07, 0.25]*, *left_pad=0.01*, *edge_padding=0.02*, *fontsize=12*, *base_string='Base'*, *limit_string='Limit'*, *format_cutoff=7*)

Rendering table of parameters which are varied during Monte Carlo analysis.

Parameters

ax
[matplotlib.axes] Axes object in which parameter table is displayed.

xpos
[float, optional] x axis position of left edge of table in axis fraction coordinates.

ypos
[float, optional] y axis position of lower edge of table in axis fraction coordinates.

height
[float, optional] Height of table in axis fraction coordinates (e.g. height = 0.5 meaning that the table has half the height of the plot).

colWidths
[list, optional] List of length 4 with widths for each column from left to right in axis fraction coordinates.

left_pad
[float, optional] Padding of the table on the left site.

edge_padding: float, optional
Padding of edges that are drawn.

fontsize
[float, optional] fontsize for table.

base_string
[str, optional] String used to label base column.

limit_string
[str, optional] String used to label limit column.

Notes

Table is rendered in provided matplotlib.axes object.

`save_results`(*file_name*)

Results of Monte Carlo simulation are saved in *file_name* and a formatted header is added. Contains name, parameter path, type and values range from *self.parameters*.

`target_price_2D_region`(*grid_points=15*)

Determining largest region spanned by first two parameters within which target prices can be achieved.

Parameters

grid_points
[int, optional] Number of grid points to determine density of grid evaluation.

Returns

self.target_price_2D_region
[dict] Dict of ndarrays with information of target price 2D region.

Notes

Model is evaluated on grid spanned by first two parameters (density of grid is controlled by grid_points), other parameters are set to limit (non-reference) values. Output is a dictionary (*self.target_price_2D_region*), which can be used to overlay target price region onto scatter ploting using plt.contourf

`target_price_components`()

Monte Carlo simulation results are sorted by H2 cost and the entries of *self.results* with a H2 cost within the specified target price range are stored in *self.target_price_data*.

pyH2A.Analysis.Monte_Carlo_Analysis.`calculate_distance`(*data*, *parameters*, *selection*, *metric='cityblock'*, *log_normalize=False*, *sum_distance=False*)

Distance of datapoints to reference is calculated using the specified metric.

Parameters

data
[ndarray] 2D array of datapoints containing parameter values for each model.

parameters
[dict] Dictionary specifying ranges for each parameter.

selection
[list] List of parameters names used for distance calculation.

metric
[str, optional] Metric used for distance calculation (e.g. 'cityblock' or 'euclidean'). Default is 'cityblock'.

log_normalize
[bool, optional] Flag to control if log normalization is used instead of linear normalization.

sum_distance
[bool, optional] Flag to control if distance is calculated by simply summing individual normalized values (equal to cityblock distance but without using absolute values, hence distance can be negative).

Returns

distances: ndarray
Array containing distance for each model.

Notes

Parameter ranges and the reference parameters are scaled to be within a n-dimensional unit cube. Distances are normalized by the number of dimensions, so that the maximum distance is always 1.

pyH2A.Analysis.Monte_Carlo_Analysis.**coordinate_position**(*x_reference*, *x_values*, *y_reference*, *y_values*)

Determine correct position for base case label in *plot_colored_scatter()*.

pyH2A.Analysis.Monte_Carlo_Analysis.**divide_into_batches**(*array*, *batch_size*)

Divide provided array into batches of size batch_size for parallel processing

pyH2A.Analysis.Monte_Carlo_Analysis.**extend_limits**(*limits_original*, *extension*)

Extend limits_original in both directions by muliplying with extensions

pyH2A.Analysis.Monte_Carlo_Analysis.**normalize_parameter**(*parameter*, *base*, *limit*, *log_normalize=False*)

Linear of log normalization of parameter (float or array) based on base and limit values.

pyH2A.Analysis.Monte_Carlo_Analysis.**select_non_reference_value**(*reference*, *values*)

Select value from values which is not the reference one.

5.5 Comparative_MC_Analysis

Classes:

Comparative_MC_Analysis(input_file)	Comparison of Monte Carlo analysis results for different models.

class pyH2A.Analysis.Comparative_MC_Analysis.**Comparative_MC_Analysis**(*input_file*)

Comparison of Monte Carlo analysis results for different models.

Parameters

Comparative_MC_Analysis > [...] > Value
[str] Path to input file for model.

Comparative_MC_Analysis > [...] > Image
[str, optional] Path to image for model.

Notes

First column of *Comparative_MC_Analysis* table can include arbitrary name for model.

Methods:

check_target_price_range_consistency()	Check that the same target price ranges are specified for all models which are to be compared.
get_models()	Get models which are to be compared from *Comparative_MC_Analysis* table in input file and perform Monte Carlo analysis for them.
plot_combined_distance([fig_width, ...])	Plot combining development distance histogram and distance/H2 cost relationship.
plot_comparative_distance_cost_relationship([...])	Plot comparative development distance/H2 cost relationship.
plot_comparative_distance_histogram([ax, ...])	Plot comparative development distance histogram.

check_target_price_range_consistency()

Check that the same target price ranges are specified for all models which are to be compared.

get_models()

Get models which are to be compared from *Comparative_MC_Analysis* table in input file and perform Monte Carlo analysis for them.

plot_combined_distance(*fig_width=12*, *fig_height=2*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, *dist_kwargs={}*, *hist_kwargs={}*, ***kwargs*)

Plot combining development distance histogram and distance/H2 cost relationship.

Parameters

fig_width
[float, optional] Width of figure in inches.

fig_height
[float, optional] Height of figure per model in inches.

target_line
[float, optional] y axis coordinate of target price line.

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

dist_kwargs: dict, optional
Dictionary containg optional keyword arguments for *plot_distance_cost_relationship()*

hist_kwargs: dict, optional
Dictionary containg optional keyword arguments for *plot_distance_histogram()*

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

plot_comparative_distance_cost_relationship(*ax=None*, *figure_lean=True*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, *dist_kwargs={}*, ***kwargs*)

Plot comparative development distance/H2 cost relationship.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

dist_kwargs: dict, optional
Dictionary containg optional keyword arguments for *plot_distance_cost_relationship()*

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[Figure_Lean object] Figure_Lean object is returned.

plot_comparative_distance_histogram(*ax=None*, *figure_lean=True*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, *hist_kwargs={}*, ***kwargs*)

Plot comparative development distance histogram.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

hist_kwargs: dict, optional
Dictionary containg optional keyword arguments for *plot_distance_histogram()*

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

5.6 Development_Distance_Time_Analysis

Classes:

Development_Distance_Time_Analysis(input_file)	Relating development distance to time using historical data.

Functions:

exponential_decline(p, x)	Exponential decay function for fitting.
fit_generic(x, y, function[, p_guess, kwargs])	Generic least squares fitting function.
linear(p, x)	Linear function for fitting.
residual_generic(p, x, y, function, **kwargs)	Generic residual function for least squares fitting.

class pyH2A.Analysis.Development_Distance_Time_Analysis.**Development_Distance_Time_Analysis**(*input_file*)

Relating development distance to time using historical data.

Parameters

Development_Distance_Time_Analysis > Input File > Value
[str] Path to textfile containing historical data for parameters specified in Monte Carlo analysis (see Notes).

Development_Distance_Time_Analysis > Log Normalization > Value
[bool] Boolean flag to control if logarithmic normalization is used for distance calculation or not. If *False*, linear normalization is used. When analyzing historical data, it is advised to use logaritthmic normalization to avoid outsized impacts of parameters which have changed on the order of magnitudes.

Development_Distance_Time_Analysis > Base Year > Value
[int] Year which corresponds to the *Base* values specified in Monte Carlo analysis.

Development_Distance_Time_Analysis > Extrapolation Limit > Year
[int] Year until which the distance/time models should be extrapolated.

Notes

The loaded textfile has to contain datapoints with historical parameter values for the specified technology. The first column contains the year of the respective datapoint and the other columns contain the corresponding parameter values. The textfile needs to be tab separated. The header of textfile has to start with # and needs to contain the tab separated column names (first being *Year* and the other being the same parameter names specified in the *Parameters - Monte_Carlo_Analysis* table.) If the textfile does not contain values for all parameters from the Monte Carlo analysis, missing parameter values are set to the base/reference value are are assumed to be constant across all years included in the textfile.

Methods:

determine_distance_time_correspondence(...)	Mapping development distance to time for plotting.
fit_historical_development_distance()	Linear and asymptotic models are fitted to historical distances and are used to extrapolate into the future.
generate_time_axis(ax, position, distances, ...)	Generating additional x axis for plot which maps development distance to time.
harmonize_monte_carlo_parameters()	Checking if all Monte Carlo parameters are included in historical dataset.
historical_development_distance()	Calculating development distance for historical and Monte Carlo analysis data.
import_data(file_name)	Importing historical data from textfile.
map_parameters()	Mapping parameters from historical data input file to Monte Carlo parameters.
plot_distance_cost_relationship([ax, ...])	Plot distance/cost relationship with additional axis mapping development distance to time.
plot_distance_histogram([ax, figure_lean, ...])	Plot distance histogram with additional axis mapping development distance to time.
plot_distance_time_relationship([ax, ...])	Ploting relationship between time and development distance based on historical data.

determine_distance_time_correspondence(*years*, *model*, *spacing=1*, *minimum_tick_distance=0.1*, *spacing_distances=array([1, 2, 3, 5, 10, 15, 20])*)

Mapping development distance to time for plotting.

Parameters

years
[ndarray] 1D array of years, for which development distances have been computed using the specified model.

model
[ndarray] 1D array of modelled, time-dependent development distance values.

spacing
[int, optional] Spacing of year ticks for visualization. Unit is years.

minimum_tick_distance
[float, optional] Minimum allowable distance between year ticks in axis coordinates (0 to 1).

spacing_distances
[ndarray, optional] Possible spacings which are to be used if specified spacing violates *minimum_tick_distance*.

Returns

labels
[ndarray] Array of year tick labels.

distances
[ndarray] Array of distances, to which year tick labels correspond.

Notes

If specified spacing leads to tick separation which is smaller than *minimum_tick_distance*, the spacing is increased by iterating through *spacing_distances* until the *minimum_tick_distance* criterion is fullfilled.

fit_historical_development_distance()

Linear and asymptotic models are fitted to historical distances and are used to extrapolate into the future.

Notes

To simply fitting of the exponential/asymptotic model, year values and distances are transformed to start at 0 and converge to 0, respectively. Both the linear and the asymptotic model are then fitted to the transformed data. The results are transformed back for visualization.

generate_time_axis(*ax*, *position*, *distances*, *labels*, *axis_label*, *color*)

Generating additional x axis for plot which maps development distance to time.

harmonize_monte_carlo_parameters()

Checking if all Monte Carlo parameters are included in historical dataset.

Notes

If Monte Carlo parameters are missing, they are added with their historical value being set to the present reference/base value.

historical_development_distance()

Calculating development distance for historical and Monte Carlo analysis data.

Notes

To ensure that historical development distances are consistent with the Monte Carlo derived development, Monte Carlo distances are recalculated using the same hyperparameters. If *log_normalize* is set to *True*, all distances are calculated using logarithmic normalization.

import_data(*file_name*)

Importing historical data from textfile.

map_parameters()

Mapping parameters from historical data input file to Monte Carlo parameters.

plot_distance_cost_relationship(*ax=None*, *figure_lean=True*, *linear_axis_y_pos=-0.25*, *expo_axis_y_pos=-0.5*, *linear_axis_label='Year (linear model)'*, *expo_axis_label='Year (asymptotic model)'*, *label_kwargs={}*, *dist_kwargs={}*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

Plot distance/cost relationship with additional axis mapping development distance to time.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

linear_axis_y_pos
[float, optional] y position of linear model time axis in axis coordinates.

expo_axis_y_pos
[float, optional] y position of exponential/asymptotic time axis in axis coordinates.

linear_axis_label
[str, optional] String for label of linear model time axis.

expo_axis_label
[str, optional] String for label of exponential/asymptotic model time axis.

label_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *determine_distance_time_correspondence()*.

dist_kwargs: dict, optional
Dictionary containg optional keyword arguments for *plot_distance_cost_relationship()*

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

`plot_distance_histogram`(*ax=None*, *figure_lean=True*, *linear_axis_y_pos=-0.4*, *expo_axis_y_pos=-0.8*, *linear_axis_label='Year (linear model)'*, *expo_axis_label='Year (asymptotic model)'*, *label_kwargs={}*, *hist_kwargs={}*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

Plot distance histogram with additional axis mapping development distance to time.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

linear_axis_y_pos
[float, optional] y position of linear model time axis in axis coordinates.

expo_axis_y_pos
[float, optional] y position of exponential/asymptotic model time axis in axis coordinates.

linear_axis_label
[str, optional] String for label of linear model time axis.

expo_axis_label
[str, optional] String for label of exponential/asymptotic model time axis.

label_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *determine_distance_time_correspondence()*.

hist_kwargs: dict, optional
Dictionary containg optional keyword arguments for *plot_distance_histogram()*

table_kwargs
[dict, optional] Dictionary containing optional keyword arguments for *render_parameter_table()*

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

`plot_distance_time_relationship`(*ax=None*, *figure_lean=True*, *legend_loc='upper left'*, *xlabel_string='Year'*, *ylabel_string='Development distance'*, *expo_label_string='Asymptotic model'*, *linear_label_string='Linear model'*, *datapoint_label_string=' historical distance'*, *markersize=10*, *color_future=True*, *parameter_table=True*, *target_distances=None*, *table_kwargs={}*, *image_kwargs={}*, *plot_kwargs={}*, ***kwargs*)

Ploting relationship between time and development distance based on historical data.

Parameters

ax
[matplotlib.axes, optional] Axes object in which plot is drawn. Default is None, creating new plot.

figure_lean
[bool, optional] If figure_lean is True, matplotlib.fig object is returned.

legend_loc
[str, optional] Controls location of legend in plot. Defaults to 'upper left'.

xlabel_string
[str, optional] String for x axis label.

ylabel_string
[str, optional] String for y axis label.

expo_label_string
[str, optional] String for label of exponential/asymptotic model.

linear_label_string
[str, optional] String for label of linear model.

datapoint_label_string
[str, optional] Second part of string for label of historical datapoints. The first part is the display name of the model.

markersize
[float, optional] Size of markers in scatter plot.

color_future
[bool, optional] Boolean flag to control if past or future region of plot is colored.

parameter_table
[bool, optional] If parameter_table is True, the parameter table is shown in the plot.

target_distances
[ndarray, optional] Target distance range as an 1D array with two entries ([lower_limit, higher_limit]), which is highlighted in the plot.

image_kwargs: dict, optional
Dictionary containing optional keyword arguments for *insert_image()*

plot_kwargs: dict, optional
Dictionary containing optional keyword arguments for *Figure_Lean()*, has priority over ***kwargs*.

****kwargs:**
Additional *kwargs* passed to *Figure_Lean()*

Returns

figure
[matplotlib.fig or None] matplotlib.fig is returned if *figure_lean* is True.

pyH2A.Analysis.Development_Distance_Time_Analysis.**exponential_decline**(*p*, *x*)

Exponential decay function for fitting.

pyH2A.Analysis.Development_Distance_Time_Analysis.**fit_generic**(*x*, *y*, *function*, *p_guess=None*, *kwargs={}*)

Generic least squares fitting function.

pyH2A.Analysis.Development_Distance_Time_Analysis.**linear**(*p*, *x*)

Linear function for fitting.

pyH2A.Analysis.Development_Distance_Time_Analysis.**residual_generic**(*p*, *x*, *y*, *function*, ***kwargs*)

Generic residual function for least squares fitting.

pyH2A is a Python framework for the analysis of hydrogen production cost.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

J

K

L

M

N

R

S

T

U

V

W