BlueSky Prototype Model

Release v1

US Energy Information Administration

Oct 24, 2024

Chapter 1

Sphinx Documentation Structure

Sphinx organizes the documentation into the following sections:

1.1 Package Overview

The documentation starts with a general overview of the main package structure. In this project, the top-level package is *src*. Inside the *src* package, you will find the following main packages:

- **electricity**: Contains modules related to electricity modeling and calculations.
- **hydrogen**: Contains modules for hydrogen energy production, storage, and consumption.
- **residential**: Contains models and utilities related to residential energy use.
- **integrator**: Integrates components from various energy sources (electricity, hydrogen, etc.) into a cohesive system.

1.2 Submodules and Subpackages

Each package may contain additional submodules and subpackages, which are organized in the documentation as follows:

- **Package**: Each package (e.g., electricity, hydrogen) is documented with a high-level description of its purpose and contents.
- **Submodules**: The individual Python modules within each package are listed and documented. For example, the *integrator* package may contain submodules like *runner.py*, *utilites.py*, and *progress_plot.py*. Each of these modules will have its own section.
- **Subpackages**: If a package contains nested subpackages, these are also documented. For instance, if *electricity* has a subpackage *scripts*, it will have its own subsection with corresponding submodules.

Each module's docstrings are captured to provide detailed information about functions, classes, methods, and attributes.

1.3 Example Structure

For the *src* package, Sphinx might organize the contents as follows:

```
src (package)/
├── integrator (subpackage)/
```

```
src (package)/
├── integrator (subpackage)/
│   ├── input
│   └── runner.py (module)
└── models (subpackage)/
    ├── electricity (package)/
    │   └── scripts (subpackage)/
    │       ├── electricity_model.py (module)
    │       └── preprocessor.py
    ├── hydrogen/
    │   ├── utilities/
    │   │   └── h2_functions.py
    │   ├── model/
    │   │   └── h2_model.py
    │   └── etc.
    └── residential/
        └── scripts/
            ├── residential.py
            └── utilites.py
```

In the HTML and Markdown outputs, each package and subpackage is represented with links **below** to the respective modules, making it easy to navigate between different sections of the documentation.

Use the **Search bar** on the top left to search for a specific function or module.

1.3.1 src

src package

Subpackages

src.integrator package

Submodules

src.integrator.config_setup module

This file contains Config_settings class. It establishes the main settings used when running the model. It takes these settings from the run_config.toml file. It contains universal configurations (e.g., configs that cut across modules and/or solve options) and module specific configs.

class `src.integrator.config_setup.`**`Config_settings`** (*config_path: Path*, *args: Namespace | None = None*, *test=False*, *years_ow=[]*, *regions_ow=[]*)

Bases: `object`

Generates the model settings that are used to solve. Settings include: - Iterative Solve Config Settings - Spatial Config Settings - Temporal Config Settings - Electricity Config Settings - Other

src.integrator.gaussseidel module

Iteratively solve 2 models with GS methodology

see README for process explanation

`src.integrator.gaussseidel.`**`run_gs`** (*settings*)

Start the iterative GS process

Parameters

settings : *obj*

Config_settings object that holds module choices and settings

src.integrator.progress_plot module

A plotter that can be used for combined solves

`src.integrator.progress_plot.`**`plot_it`** (*h2_price_records=[]*, *elec_price_records=[]*, *h2_obj_records=[]*, *elec_obj_records=[]*, *h2_demand_records=[]*, *elec_demand_records=[]*, *load_records=[]*, *elec_price_to_res_records=[]*)

cheap plotter of iterative progress

`src.integrator.progress_plot.`**`plot_price_distro`** (*price_records: list[float]*)
cheap/quick analyisis and plot of the price records

src.integrator.runner module

A gathering of functions for running models solo

`src.integrator.runner.`**`run_elec_solo`** (*settings: Config_settings | None = None*)
Runs electricity model by itself as defined in settings

Parameters

settings: Config_settings
Contains configuration settings for which regions, years, and switches to run

`src.integrator.runner.`**`run_h2_solo`** (*settings: Config_settings | None = None*)
Runs hydrogen model by itself as defined in settings

Parameters

settings: Config_settings
Contains configuration settings for which regions and years to run

`src.integrator.runner.`**`run_residential_solo`** (*settings: Config_settings | None = None*)
Runs residential model by itself as defined in settings

Parameters

settings: Config_settings
Contains configuration settings for which regions and years to run

`src.integrator.runner.`**`run_standalone`** (*settings: Config_settings*)
Runs standalone methods based on settings selections; running 1 or more modules

Parameters

settings : *Config_settings*
Instance of config_settings containing run options, mode and settings

src.integrator.unified module

Unifying the solve of both H2 and Elec and Res

Dev Notes:

(1). The "annual demand" constraint that is present and INACTIVE

is omitted here for clarity. It may likely be needed–in some form–at a later

time. Recall, the key linkages to share the electrical demand primary variable are:

(a). an annual level demand constraint (b). an accurate price-pulling function that can consider weighted duals

from both constraints [NOT done]

(2). This model has a 2-solve update cycle as commented on near the termination check

elec_prices gleaned from cycle[n] results -> solve cycle[n+1] new_load gleaned from cycle[n+1] results -> solve cycle[n+2] elec_pices gleaned from cycle[n+2]

`src.integrator.unified.`**`run_unified`** (*settings: Config_settings*)
Runs unified solve method based on

Parameters

settings : *Config_settings*
Instance of config_settings containing run options, mode and settings

src.integrator.utilities module

A gathering of utility functions for dealing with model interconnectivity

Dev Note: At some review point, some decisions may move these back & forth with parent models after it is decided if it is a utility job to do …. or a class method.

Additionally, there is probably some renaming due here for consistency

class `src.integrator.utilities.`**`EI`** (*region*, *year*, *hour*)

Bases: `tuple`

(region, year, hour)

`hour`

Alias for field number 2

`region`

Alias for field number 0

`year`

Alias for field number 1

class `src.integrator.utilities.`**`HI`** (*region*, *year*)

Bases: `tuple`

(region, year)

`region`

Alias for field number 0

`year`

Alias for field number 1

`src.integrator.utilities.`**`convert_elec_price_to_lut`** (*prices: list[tuple[EI, float]]*) → dict[EI, float]

convert electricity prices to dictionary, look up table

Parameters

prices : *list[tuple[EI, float]]*

list of prices

Returns

dict[EI, float]

dict of prices

`src.integrator.utilities.`**`convert_h2_price_records`** (*records: list[tuple[HI, float]]*) → dict[HI, float]

simple coversion from list of records to a dictionary LUT repeat entries should not occur and will generate an error

`src.integrator.utilities.`**`create_temporal_mapping`** (*sw_temporal*)

Combines the input mapping files within the electricity model to create a master temporal mapping dataframe. The df is used to build multiple temporal parameters used within the model. It creates a single dataframe that has 8760 rows for each hour in the year. Each hour in the year is assigned a season type, day type, and hour type used in the model. This defines the number of time periods the model will use based on cw_s_day and cw_hr inputs.

Returns

dataframe

a dataframe with 8760 rows that include each hour, hour type, day, day type, and season. It also includes the weights for each day type and hour type.

`src.integrator.utilities.`**`get_annual_wt_avg`** (*elec_price: DataFrame*) → dict[HI, float]

takes annual weighted average of hourly electricity prices

Parameters

elec_price : *pd.DataFrame*

hourly electricity prices

Returns

dict[HI, float]
annual weighted average electricity prices

`src.integrator.utilities.`**`get_elec_price`** (*instance: PowerModel | ConcreteModel*, *block=None*) → DataFrame

pulls hourly electricity prices from completed PowerModel and de-weights them

Prices from the duals are weighted by the day and year weights applied in the OBJ function This function retrieves the prices for all hours and removes the day and annual weights to return raw prices (and the day weights to use as needed)

Parameters

instance : *PowerModel*
solved electricity model

block: ConcreteModel
reference to the block if the electricity model is a block within a larger model

Returns

pd.DataFrame
df of raw prices and the day weights to re-apply (if needed) columns: [r, y, hour, day_weight, raw_price]

`src.integrator.utilities.`**`get_output_root`** ( )

get the name of the output dir, which includes the name of the mode type and a timestamp

Returns

path
output directory path

`src.integrator.utilities.`**`make_dir`** (*dir_name*)

generates an output directory to write model results, output directory is the date/time at the time this function executes. It includes subdirs for vars, params, constraints.

Returns

string
the name of the output directory

`src.integrator.utilities.`**`poll_h2_demand`** (*model: PowerModel*) → dict[HI, float]

Get the hydrogen demand by rep_year and region

Use the Generation variable for h2 techs

NOTE: Not sure about day weighting calculation here!!

Returns

dict[HI, float]
dictionary of prices by H2 Index: price

`src.integrator.utilities.`**`poll_h2_prices_from_elec`** (*model: PowerModel*, *tech*, *regions: Iterable*) → dict[Any, float]

poll the step-1 H2 price currently in the model for region/year, averaged over any steps

`src.integrator.utilities.`**`poll_hydrogen_price`** (*model: H2Model | ConcreteModel*, *block=None*) → list[tuple[HI, float]]

Retrieve the price of H2 from the H2 model

Parameters

model : *H2Model*
the model to poll

block: optional
block model to poll

Returns

list[tuple[HI, float]]
list of H2 Index, price tuples

`src.integrator.utilities.`**`poll_year_avg_elec_price`** (*price_list: list[tuple[EI, float]]*) → dict[HI, float]

retrieve a REPRESENTATIVE price at the annual level from a listing of prices

This function computes the AVERAGE elec price for each region-year combo

Parameters

price_list : *list[tuple[EI, float]]*
input price list

Returns

dict[HI, float]
a dictionary of (region, year): price

`src.integrator.utilities.`**`regional_annual_prices`** (*m: PowerModel | ConcreteModel, block=None*) → dict[HI, float]

pulls all regional annual weighted electricity prices

Parameters

m : *typing.Union['PowerModel', ConcreteModel]*
solved PowerModel

block : *optional*
solved block model if applicable, by default None

Returns

dict[HI, float]
dict with regional annual electricity prices

`src.integrator.utilities.`**`select_solver`** (*instance: ConcreteModel*)

Select solver based on learning method

Parameters

instance : *PowerModel*
electricity pyomo model

Returns

solver type (?)
The pyomo solver

`src.integrator.utilities.`**`setup_logger`** (*output_dir*)

initiates logging, sets up logger in the output directory specified

Parameters

output_dir : *path*
output directory path

`src.integrator.utilities.`**`simple_solve`** (*m: ConcreteModel*)

a simple solve routine

`src.integrator.utilities.`**`simple_solve_no_opt`** (*m: ~pyomo.core.base.PyomoModel.ConcreteModel, opt: <pyomo.opt.base.solvers.SolverFactoryClass object at 0x00000295031CDE10>*)

Solve concrete model using solver factory object

Parameters

m : *ConcreteModel*
Pyomo model

opt: SolverFactory
Solver object initiated prior to solve

`src.integrator.utilities.`**`update_elec_demand`** (*self, elec_demand: dict[HI, float]*) → None

Update the external electical demand parameter with demands from the H2 model

Parameters

elec_demand : *dict[HI, float]*
: the new demands broken out by hyd index (region, year)

`src.integrator.utilities.`**`update_h2_prices`** (*model: PowerModel*, *h2_prices: dict[HI, float]*) → None

Update the H2 prices held in the model

Parameters

h2_prices : *list[tuple[HI, float]]*
: new prices

Module contents

src.models package

Subpackages

src.models.electricity package

Subpackages

src.models.electricity.scripts package

Subpackages

src.models.electricity.scripts.common package

Submodules

src.models.electricity.scripts.common.common module

Utility file containing miscellaneous common functions

`src.models.electricity.scripts.common.common.`**`check_results`** (*results*, *SolutionStatus*, *TerminationCondition*)

Check results for termination condition and solution status

Parameters

results : *str*
: Results from pyomo

SolutionStatus : *str*
: Solution Status from pyomo

TerminationCondition : *str*
: Termination Condition from pyomo

Returns

results

Module contents

Submodules

src.models.electricity.scripts.electricity_model module

Electricity Model. This file contains the PowerModel class which contains a pyomo optimization model of the electric power sector. The class is organized by sections: settings, sets, parameters, variables, objective function, constraints, plus additional misc support functions.

class `src.models.electricity.scripts.electricity_model.`**`PowerModel`** (**args*, ***kwds*)

Bases: `ConcreteModel`

A PowerModel instance. Builds electricity pyomo model.

Parameters

all_frames : *dictionary of pd.DataFrames*
: Contains all dataframes of inputs

setA : *Sets*
: Contains all other non-dataframe inputs

src.models.electricity.scripts.postprocessor module

This file is the main postprocessor for the electricity model. It writes out all relevant model outputs (e.g., variables, sets, parameters, constraints). It contains:

- A function that converts pyomo component objects to dataframes
- A function that writes the dataframes to output directories
- A function to make the electricity output sub-directories
- The postprocessor function, which loops through the model component objects and applies the

functions to convert and write out the data to dfs to the electricity output sub-directories

`src.models.electricity.scripts.postprocessor.`**`make_elec_output_dir`**( )

generates an output directory to write model results, output directory is the date/time at the time this function executes. It includes subdirs for vars, params, constraints.

Returns

string
the name of the output directory

`src.models.electricity.scripts.postprocessor.`**`postprocessor`**(*instance*)

master postprocessor function that writes out the final dataframes from to the electricity model. Creates the output directories and writes out dataframes for variables, parameters, and constraints. Gets the correct columns names for each dataframe using the cols_dict.

Parameters

instance : *pyomo model*
electricity concrete model

Returns

string
output directory name

`src.models.electricity.scripts.postprocessor.`**`report_obj_df`**(*mod_object*, *instance*, *dir_out*, *sub_dir*)

Creates a df of the component object within the pyomo model, separates the key data into different columns and then names the columns if the names are included in the cols_dict. Writes the df out to the output directory.

Parameters

obj : *pyomo component object*
e.g., pyo.Var, pyo.Set, pyo.Param, pyo.Constraint

instance : *pyomo model*
electricity concrete model

dir_out : *str*
output electricity directory

sub_dir : *str*
output electricity sub-directory

src.models.electricity.scripts.preprocessor module

This file is the main preprocessor for the electricity model. It established the parameters and sets that will be used in the model. It contains:

- A class that contains all sets used in the model
- A collection of support functions to read in and setup parameter data
- The preprocessor function, which produces an instance of the Set class and a dict of params
- A collection of support functions to write out the inputs to the output directory

class `src.models.electricity.scripts.preprocessor.`**`Sets`**(*settings*)

Bases: `object`

Generates an initial batch of sets that are used to solve electricity model. Sets include: - Scenario descriptor and model switches - Regional sets - Temporal sets - Technology type sets - Supply curve step sets - Other

`src.models.electricity.scripts.preprocessor.`**`add_season_index`** (*cw_temporal*, *df*, *pos*)

adds a season index to the input dataframe

Parameters

cw_temporal : *dataframe*
dataframe that includes the season index

df : *dataframe*
parameter data to be modified

pos : *int*
column position for the seasonal set

Returns

dataframe
modified parameter data now indexed by season

`src.models.electricity.scripts.preprocessor.`**`avg_by_group`** (*df*, *set_name*, *map_frame*)

takes in a dataframe and groups it by the set specified and then averages the data.

Parameters

df : *dataframe*
parameter data to be modified

set_name : *str*
name of the column/set to average the data by

map_frame : *dataframe*
data that maps the set name to the new grouping for that set

Returns

dataframe
parameter data that is averaged by specified set mapping

`src.models.electricity.scripts.preprocessor.`**`fill_values`** (*row*, *subset_list*)

Function to fill in the subset values, is used to assign all years within the year solve range to each year the model will solve for.

Parameters

row : *int*
row number in df

subset_list : *list*
list of values to map

Returns **int** value from subset_list

`src.models.electricity.scripts.preprocessor.`**`load_data`** (*tablename*, *metadata*, *engine*)

loads the data from the SQL database; used in readin_sql function.

Parameters

tablename : *string*
table name

metadata : *SQL metadata*
SQL metadata

engine : *SQL engine*
SQL engine

Returns

dataframe
table from SQL db as a dataframe

`src.models.electricity.scripts.preprocessor.`**`makedir`** (*dir_out*)
creates a folder directory based on the path provided

Parameters

dir_out : *str*
path of directory

`src.models.electricity.scripts.preprocessor.`**`output_inputs`** ( )
function developed initial for QA purposes, writes out to csv all of the dfs and sets passed to the electricity model to an output directory.

Returns

all_frames : *dictionary*
dictionary of dataframes where the key is the file name and the value is the table data

setin : *Sets*
an initial batch of sets that are used to solve electricity model

`src.models.electricity.scripts.preprocessor.`**`preprocessor`** (*setin*)
master preprocessor function that generates the final dataframes and sets sent over to the electricity model. This function reads in the input data, modifies it based on the temporal and regional mapping specified in the inputs, and gets it into the final formatting needed. Also adds some additional regional sets to the set class based on parameter inputs.

Parameters

setin : *Sets*
an initial batch of sets that are used to solve electricity model

Returns

all_frames : *dictionary*
dictionary of dataframes where the key is the file name and the value is the table data

setin : *Sets*
an initial batch of sets that are used to solve electricity model

`src.models.electricity.scripts.preprocessor.`**`print_sets`** (*setin*)
function developed initially for QA purposes, prints out all of the sets passed to the electricity model.

Parameters

setin : *Sets*
an initial batch of sets that are used to solve electricity model

`src.models.electricity.scripts.preprocessor.`**`readin_csvs`** (*all_frames*)
Reads in all of the CSV files from the input dir and returns a dictionary of dataframes, where the key is the file name and the value is the table data.

Parameters

all_frames : *dictionary*
empty dictionary to be filled with dataframes

Returns

dictionary
completed dictionary filled with dataframes from the input directory

`src.models.electricity.scripts.preprocessor.`**`readin_sql`** (*all_frames*)
Reads in all of the tables from a SQL databased and returns a dictionary of dataframes, where the key is the table name and the value is the table data.

Parameters

all_frames : *dictionary*
empty dictionary to be filled with dataframes

Returns

dictionary
completed dictionary filled with dataframes from the input directory

`src.models.electricity.scripts.preprocessor.`**`subset_dfs`** (*all_frames*, *setin*, *i*)

filters dataframes based on the values within the set

Parameters

all_frames : *dictionary*
dictionary of dataframes where the key is the file name and the value is the table data

setin : *Sets*
contains an initial batch of sets that are used to solve electricity model

i : *string*
name of the set contained within the sets class that the df will be filtered based on.

Returns

dictionary
completed dictionary filled with dataframes filtered based on set inputs specified

src.models.electricity.scripts.runner module

This file is a collection of functions that are used to build, run, and solve the electricity model.

`src.models.electricity.scripts.runner.`**`build_elec_model`** (*all_frames*, *setin*) → PowerModel

building pyomo electricity model

Parameters

all_frames : *dict of pd.DataFrame*
input data frames

setin : *Sets*
input settings Sets

Returns

PowerModel
built (but unsolved) electricity model

`src.models.electricity.scripts.runner.`**`cost_learning_func`** (*instance*, *pt*, *y*)

function for updating learning costs by technology and year

Parameters

instance : *PowerModel*
electricity pyomo model

pt : *int*
technology type

y : *int*
year

Returns **int** updated capital cost based on learning calculation

`src.models.electricity.scripts.runner.`**`init_old_cap`** (*instance*)

initialize capacity for 0th iteration

Parameters

instance : *PowerModel*
unsolved electricity model

`src.models.electricity.scripts.runner.`**`run_elec_model`** (*settings*, *solve=True*) → PowerModel

build electricity model (and solve if solve=True) after passing in settings

Parameters

settings : *Config_settings*
Configuration settings

solve : *bool, optional*
solve electricity model?, by default True

Returns

PowerModel
electricity model

`src.models.electricity.scripts.runner.`**`set_new_cap`** (*instance*)

calculate new capacity after solve iteration

Parameters

instance : *PowerModel*
solved electricity pyomo model

`src.models.electricity.scripts.runner.`**`solve_elec_model`** (*instance*)

solve electicity model

Parameters

instance : *PowerModel*
built (but not solved) electricity pyomo model

`src.models.electricity.scripts.runner.`**`update_cost`** (*instance*)

update capital cost based on new capacity learning

Parameters

instance : *PowerModel*
electricity pyomo model

src.models.electricity.scripts.utilities module

This file is a collection of functions that are used in support of the electricity model.

`src.models.electricity.scripts.utilities.`**`annual_count`** (*hour*, *m*) → int

return the aggregate weight of this hour in the representative year we know the hour weight, and the hours are unique to days, so we can get the day weight

Parameters

hour : *int*
the rep_hour

Returns **int** the aggregate weight (count) of this hour in the rep_year. NOT the hour weight!

`src.models.electricity.scripts.utilities.`**`create_obj_df`** (*mod_object*)

takes pyomo component objects (e.g., variables, parameters, constraints) and processes the pyomo data and converts it to a dataframe and then writes the dataframe out to an output dir. The dataframe contains a key column which is the original way the pyomo data is structured, as well as columns broken out for each set and the final values.

Parameters

mod_object : *pyomo component object*
pyomo component object

Returns

pd.DataFrame
contains the pyomo model results for the component object

`src.models.electricity.scripts.utilities.`**`declare_param`** (*self*, *pname*, *p_set*, *data*, *default=0*, *mutable=False*)

Assigns the df to be a pyomo parameter using the name specified. Adds the name and index column names to the column dictionary used for post-processing.

Parameters

pname : *string*
name of the parameter to be declared

p_set : *pyomo set*
the pyomo set that cooresponds to the parameter data

data : *dataframe, series, float, or int*
dataframe used generate the parameter

default : *int, optional*
by default 0

mutable : *bool, optional*
by default False

Returns

pyomo parameter
a pyomo parameter

`src.models.electricity.scripts.utilities.`**`declare_set`** (*self*, *sname*, *df*)

Assigns the index from the df to be a pyomo set using the name specified. Adds the name and index column names to the column dictionary used for post-processing.

Parameters

sname : *string*
name of the set to be declared

df : *dataframe*
dataframe from which the index will be grabbed to generate the set

Returns

pyomo set
a pyomo set

`src.models.electricity.scripts.utilities.`**`declare_var`** (*self*, *vname*, *v_set*, *bound=(0, 1000000000)*)

Assigns the set to be the index for the pyomo variable being declared. Adds the name and index column names to the column dictionary used for post-processing.

Parameters

vname : *str*
name of pyomo variable

v_set : *pyomo set*
the pyomo set that the variable data will be indexed by

bound : *set, optional*
optional argument for setting variable bounds, default values set to zero to one billion

Returns

pyomo variable
a pyomo variable

`src.models.electricity.scripts.utilities.`**`populate_RM_sets_rule`** (*m*)

Creates new reindexed sets for reserve margin constraint

Parameters

m : *PowerModel*
pyomo electricity model instance

`src.models.electricity.scripts.utilities.`**`populate_by_hour_sets_rule`** (*m*)

Creates new reindexed sets for dispatch_cost calculations

Parameters

m : *PowerModel*
pyomo electricity model instance

`src.models.electricity.scripts.utilities.`**`populate_demand_balance_sets_rule`**(*m*)

Creates new reindexed sets for demand balance constraint

Parameters

m : *PowerModel*
pyomo electricity model instance

`src.models.electricity.scripts.utilities.`**`populate_hydro_sets_rule`**(*m*)

Creates new reindexed sets for hydroelectric generation seasonal upper bound constraint

Parameters

m : *PowerModel*
pyomo electricity model instance

`src.models.electricity.scripts.utilities.`**`populate_reserves_sets_rule`**(*m*)

Creates new reindexed sets for operating reserves constraints

Parameters

m : *PowerModel*
pyomo electricity model instance

`src.models.electricity.scripts.utilities.`**`populate_sets_rule`**(*m1*, *sname*, *set_base_name=''*, *set_base2=[]*)

Generic function to create a new re-indexed set for a PowerModel instance which should speed up build time. Must pass non-empty (either) set_base_name or set_base2

Parameters

m1 : *PowerModel*
electricity pyomo model instance

sname : *str*
name of input pyomo set to base reindexing

set_base_name : *str, optional*
the name of the set to be the base of the reindexing, if left blank, uses set_base2, by default ''

set_base2 : *list, optional*
the list of names of set columns to be the base of the reindexing, if left blank, should use set_base_name, by default []

Returns

pyomo set
reindexed set to be added to electricity model

`src.models.electricity.scripts.utilities.`**`populate_trade_sets_rule`**(*m*)

Creates new reindexed sets for trade constraints

Parameters

m : *PowerModel*
pyomo electricity model instance

Module contents

Module contents

src.models.hydrogen package

Subpackages

src.models.hydrogen.model package

Submodules

src.models.hydrogen.model.actions module

A sequencer for actions in the model. This may change up a bit, but it is a place to assert control of the execution sequence for now

`src.models.hydrogen.model.actions.`**`build_grid`** (*grid_data: GridData*) → Grid

build a grid from grid_data

Parameters

grid_data: obj
GridData object to build grid from

Returns

Grid : *obj*
Grid object

`src.models.hydrogen.model.actions.`**`build_model`** (*grid: Grid*, ***kwds*) → H2Model

build model from grd

Parameters

grid : *obj*
Grid object to build model from

Returns

H2Model : *obj*
H2Model object

`src.models.hydrogen.model.actions.`**`load_data`** (*path_to_input: Path*, ***kwds*) → GridData

load data for model

Parameters

path_to_input : *Path*
Data folder path

Returns

GridData : *obj*
Grid Data object from path

`src.models.hydrogen.model.actions.`**`make_h2_outputs`** (*model*)

save model outputs

Parameters

model : *obj*
Solved H2Model

`src.models.hydrogen.model.actions.`**`quick_summary`** (*solved_hm: H2Model*) → None

print and return summary of solve

Parameters

solved_hm : *obj*
Solved H2Model

Returns

res : *str*
Printed summary

`src.models.hydrogen.model.actions.`**`run_hydrogen_model`** (*settings*)

run hydrogen model in standalone

Parameters

settings : *obj*
Config_setup instance

`src.models.hydrogen.model.actions.`**`solve_it`** (*hm: H2Model*) → SolverResults

solve hm

Parameters

hm : *objH2Model*
H2Model to solve

Returns

SolverResults : *obj*
results of solve

src.models.hydrogen.model.h2_model module

The Hydrogen Model takes an a Grid object and uses it to populate a Pyomo model that solves for the least cost to produce and distribute Hydrogen by electrolysis across the grid to satisfy a given demand, returning the duals as shadow prices. It can be run in stand-alone or integrated runs. If stand-alone, a function for generated temporally varying data must be supplied. By default it simply projects geometric growth for electricity price and demand.

class `src.models.hydrogen.model.h2_model.`**`H2Model`** (**args*, ***kwds*)

Bases: `ConcreteModel`

`poll_electric_demand` ( ) → dict[HI, float]

compute the electrical demand by region-year after solve
Note: we will use production * 1/eff to compute electrical demand

Parameters

hm : *H2Model*
self

Returns

dict[HI, float]
electricity demand by region, year. (region, year):demand

`update_exchange_params` (*new_demand=None*, *new_electricity_price=None*)

update exchange parameters in integrated mode

Parameters

hm : *H2Model*
model

new_demand : *dict, optional*
new demand (region, year):value. Defaults to None.

new_electricity_price : *dict, optional*
new electricity prices (region,year):value . Defaults to None.

`src.models.hydrogen.model.h2_model.`**`resolve`** (*hm: H2Model*, *new_demand=None*, *new_electricity_price=None*, *test=False*)

For convenience: After building and solving the model initially:

if you want to solve without annual data by applying a geometric growth rate to exhcange parameters

Parameters

hm : *H2Model*
model

new_demand : *dict, optional*
new_demand[region,year] for H2demand in (region,year). Defaults to None.

new_electricity_price : *dict, optional*
new_electricity_price[region,year]. Defaults to None.

test : *bool, optional*
is this just a test? Defaults to False.

`src.models.hydrogen.model.h2_model.`**`solve`** (*hm: H2Model*)

summary

Parameters

hm : *H2Model*
self

Raises

RuntimeError
no optimal solution to problem

src.models.hydrogen.model.validators module

set of validator functions for use in model

`src.models.hydrogen.model.validators.`**`region_validator`** (*hm: H2Model*, *region*)

checks if region name is string or numeric

Parameters

hm : *H2Model*

model

region : *any*
region name

Raises:
ValueError: region wrong type

Returns:
bool: is correct type

Module contents

src.models.hydrogen.network package

Submodules

src.models.hydrogen.network.grid module

GRID CLASS

This is the central class that binds all the other classes together. No class instance exists in a reference that isn't fundamentally contained in a grid. The grid is used to instantiate a model, read data, create the regionality and hub / arc network within that regionality, assign data to objects and more.

notably, the grid is used to coordinate internal methods in various classes to make sure that their combined actions keep the model consistent and accomplish the desired task.

class `src.models.hydrogen.network.grid.`**`Grid`** (*data: GridData* | *None = None*)

Bases: `object`

`aggregate_hubs` (*hublist*, *region*)

combine all hubs in hublist into a single hub, and place them in region. Arcs that connect to any of these hubs also get aggegated into arcs that connect to the new hub
and their original origin / destination that's not in hublist.

Parameters

hublist : *list*
list of hubs to aggregate

region : *Region*
region to place them in

`arc_generation` (*df*)

generate arcs from the arc data

Parameters

df : *DataFrame*
arc data

build_grid (*vis=True*)

builds a grid fom the GridData by recursively adding regions starting at top-level region 'world'.

Parameters

vis : *bool, optional*
if True, will generate an image of the hub-network with regional color-coding. Defaults to True.

collapse (*region_name*)

make a region absorb all it's sub-regions and combine all its and its childrens hubs into one

Parameters

region_name : *str*
region to collapse

collapse_level (*level*)

collapse all regions at a specific level of depth in the regional hierarchy, with world = 0

Parameters

level : *int*
level to collapse

combine_arcs (*arclist, origin, destination*)

combine a set of arcs into a single arc with given origin and destination

Parameters

arclist : *list*
list of arcs to aggregate

origin : *str*
new origin hub

destination : *str*
new destination hub

connect_subregions ( )

create an arc for all hubs in bottom-level regions to whatever hub is located in their parent region

create_arc (*origin, destination, capacity, cost=0.0*)

Creates and arc from origin to destination with given capacity and cost

Parameters

origin : *str*
origin hub name

destination : *str*
destination hub name

capacity : *float*
capacity of arc

cost : *float, optional*
cost of transporting 1kg H2 along arc. Defaults to 0.

create_hub (*name, region, data=None*)

creates a hub in a given region

Parameters

name : *str*
hub name

region : *Region*
Region hub is placed in

data : *DataFrame, optional*
dataframe of hub data to append. Defaults to None.

create_region (*name*, *parent=None*, *data=None*)
creates a region with a given name, parent region, and data
Parameters
name : *str*
name of region
parent : *Region, optional*
parent region. Defaults to None.
data : *DataFrame, optional*
region data. Defaults to None.

delete (*thing*)
deletes a hub, arc, or region
Parameters
thing : *Hub, Arc, or Region*
thing to delete

load_hubs ( )
load hubs from data

recursive_region_generation (*df*, *parent*)
cycle through a region dataframe, left column to right until it hits data column, adding new regions and subregions according to how it is hierarchically structured.
Future versions should implement this with a graph structure for the data instead of a dataframe, which would be more natural.
Parameters
df : *DataFrame*
hierarchically structured dataframe of regions and their data.
parent : *Region*
Parent region

test ( )
test run

visualize ( )
visualize the grid network using graphx

write_data ( )
_write data to file

src.models.hydrogen.network.grid_data module

grid_data is the the data object that grids are generated from. It reads in raw data with a region filter, and holds it in one structure for easy access

class `src.models.hydrogen.network.grid_data.`**`GridData`** (*data_folder: Path*, *regions_of_interest: list[str] | None = None*)
Bases: `object`

src.models.hydrogen.network.hub module

HUB CLASS

class objects are individual hubs, which are fundamental units of production in the model. Hubs belong to regions, and connect to each other with transportation arcs.

class `src.models.hydrogen.network.hub.`**`Hub`** (*name, region, data=None*)

Bases: `object`

`add_inbound` (*arc*)

add an inbound arc to hub

Parameters

arc : *Arc*

add an inbound arc to hub

`add_outbound` (*arc*)

add an outbound arc to hub

Parameters

arc : *Arc*

arc to add

`change_region` (*new_region*)

move hub to new region

Parameters

new_region : *Region*

region hub should be moved to

`cost` (*technology, year*)

return a cost value in terms of data fields

Parameters

technology : *str*

technology type

year : *int*

year

Returns

float

a cost value

`display_outbound` ( )

print all outbound arcs from hub

`get_data` (*quantity*)

fetch quantity from hub data

Parameters

quantity : *str*

name of data field to fetch

Returns

float or str

quantity to be fetched

`remove_inbound` (*arc*)

remove an inbound arc from hub

Parameters

arc : *Arc*

arc to remove

`remove_outbound` (*arc*)

remove an outbound arc from hub

Parameters

arc : *Arc*

arc to remove

src.models.hydrogen.network.region module

Region class:

Class objects are regions, which have a natural tree-structure. Each region can have a parent region and child regions (subregions), a data object, and a set of hubs.

class `src.models.hydrogen.network.region.`**`Region`** (*name*, *grid=None*, *kind=None*, *data=None*, *parent=None*)

Bases: `object`

`absorb_subregions` ( )
delete subregions, acquire their hubs and subregions

`absorb_subregions_deep` ( )
absorb subregions recursively so that region becomes to the deepest level in the hierarchy

`add_hub` (*hub*)
add a hub to region

Parameters

hub : *Hub*
hub to add

`add_subregion` (*subregion*)
make a region a subregion of self

Parameters

subregion : *Region*
new subregion

`aggregate_subregion_data` (*subregions*)
combine the data from subregions and assign it to self

Parameters

subregions : *list*
list of subregions

`assigned_names` = {}

`create_subregion` (*name*, *data=None*)
create a subregion

Parameters

name : *str*
subregion name

data : *DataFrame, optional*
subregion data. Defaults to None.

`delete` ( )
delete self, reassign hubs to parent, reassign children to parent

`display_children` ( )
display child regions

`display_hubs` ( )
display hubs

`get_data` (*quantity*)
pull data from region data

Parameters

quantity : *str*
name of data field in region data

Returns

str, float
value of data

remove_hub (*hub*)
remove hub from region

Parameters

hub : *Hub*
hub to remove

remove_subregion (*subregion*)
remove a subregion from self

Parameters

subregion : *Region*
subregion to remove

update_data (*df*)
change region data

Parameters

df : *DataFrame*
new data

update_parent (*new_parent*)
change parent region

Parameters

new_parent : *Region*
new parent region

src.models.hydrogen.network.registry module

REGISTRY CLASS

This class is the central registry of all objects in a grid. It preserves them in dicts of object-name:object so that they can be looked up by name. it also should serve as a place to save data in different configurations for faster parsing - for example, depth is a dict that organizes regions according to their depth in the region nesting tree.

class `src.models.hydrogen.network.registry.`**Registry**
Bases: `object`

add (*thing*)
add a thing to the registry. Thing can be Hub,Arc, or Region

Parameters

thing : *Arc, Region, or Hub*
thing to add to registry

Returns

Arc, Region, or Hub
thing being added gets returned

remove (*thing*)
remove thing from registry

Parameters

thing : *Arc, Hub, or Region*
thing to remove

update_levels ( )
update dictionary of regions by level

src.models.hydrogen.network.transportation_arc module

TRANSPORTATION ARC CLASS

objects in this class represent individual transportation arcs. An arc can exist with zero capacity, so they only represent *possible* arcs.

class `src.models.hydrogen.network.transportation_arc.`**`TransportationArc`** (*origin*, *destination*, *capacity*, *cost=0*)

Bases: `object`

`change_destination` (*new_destination*)

change the destination hub of arc

Parameters

new_destination : *Hub*

new destination hub

`change_origin` (*new_origin*)

change the origin hub of arc

Parameters

new_origin : *Hub*

new origin hub

`disconnect` ( )

disconnect arc from it's origin and destination

Module contents

src.models.hydrogen.utilities package

Submodules

src.models.hydrogen.utilities.h2_functions module

`src.models.hydrogen.utilities.h2_functions.`**`get_demand`** (*hm: H2Model*, *region*, *time*)

get demand for region at time. If mode not standard, just increase demand by 5% per year

Parameters

hm : *H2Model*

model

region : *str*

region

time : *int*

year

Returns

float

demand

`src.models.hydrogen.utilities.h2_functions.`**`get_elec_price`** (*hm: H2Model*, *region*, *year*)

get electricity price in region, year Parameters ———- hm : H2Model

_model

region : *str*

region

year : *int*

year

Returns

float

electricity price in region and year

`src.models.hydrogen.utilities.h2_functions.`**`get_electricity_consumption_rate`** (*hm: H2Model*, *tech*)

the electricity consumption rate for technology type tech Parameters ———- hm : H2Model

model

tech : *str*

technology type

Returns

float

GWh per kg H2

`src.models.hydrogen.utilities.h2_functions.`**`get_electricty_consumption`** (*hm: H2Model*, *region*, *year*)

get electricity consumption for region, year Parameters ———- hm : H2Model

model

region : *str*

region

year : *int*

year

Returns

float

the elecctricity consumption for a region and year in the model

`src.models.hydrogen.utilities.h2_functions.`**`get_gas_price`** (*hm: H2Model*, *region*, *year*)

get gas price for region, year

Parameters

hm : *H2Model*

model

region : *str*

region

year : *int*

year

Returns

float

gas price in region and year

`src.models.hydrogen.utilities.h2_functions.`**`get_production_cost`** (*hm: H2Model*, *hub*, *tech*, *year*)

return production cost for tech at hub in year

Parameters

hm : *H2Model*

model

hub : *str*

hub

tech : *str*

technology type

year : *int*

year

Returns

float

production cost of H2 for tech at hub in year

Module contents

Module contents

src.models.residential package

Subpackages

src.models.residential.preprocessor package

Submodules

src.models.residential.preprocessor.enduse_db module

src.models.residential.preprocessor.enduse_demand module

src.models.residential.preprocessor.generate_inputs module

This file contains the options to re-create the input files. It creates:

- Load.csv: electricity demand for all model years (used in residential and electricity)
- BaseElecPrice.csv: electricity prices for initial model year (used in residential only)

Uncomment out the functions at the end of this file in the "if __name__ == '__main__'" statement in order to generate new load or base electricity prices.

`src.models.residential.preprocessor.generate_inputs.`**`base_price`**`( )`

Runs the electricity model with base price configuration settings and then merges the electricity prices and temporal crosswalk data produced from the run to generate base year electricity prices.

Returns

pandas.core.frame.DataFrame

dataframe that contains base year electricity prices for all regions/hours

`src.models.residential.preprocessor.generate_inputs.`**`compare_load_method_results`**`( )`

runs the two methods for developing future load estimates and then creates to review files. review1 sums the hourly data up by region and year. review2 writes out the hourly data for the final model year for all regions. The data is written out to csvs for user inspection.

`src.models.residential.preprocessor.generate_inputs.`**`scale_load`**`( )`

Reads in BaseLoad.csv (load for all regions/hours for first year) and LoadScalar.csv (a multiplier for all model years). Merges the data and multiplies the load by the scalar to generate new load estimates for all model years.

Returns

pandas.core.frame.DataFrame

dataframe that contains load for all regions/years/hours

`src.models.residential.preprocessor.generate_inputs.`**`scale_load_with_enduses`**`( )`

Reads in BaseLoad.csv (load for all regions/hours for first year), EnduseBaseShares.csv (the shares of demand for each enduse in the base year) and EnduseScalar.csv (a multiplier for all model years by enduse category). Merges the data and multiplies the load by the adjusted enduse scalar and then sums up to new load estimates for all model years.

Returns

pandas.core.frame.DataFrame

dataframe that contains load for all regions/years/hours

Module contents

src.models.residential.scripts package

Submodules

src.models.residential.scripts.residential module

Residential Model. This file contains the residentialModule class which contains a representation of residential electricity prices and demands.

class `src.models.residential.scripts.residential.`**`residentialModule`** (*settings: Config_settings | None = None, loadFile: str | None = None, load_df: DataFrame | None = None, calibrate: bool | None = False*)

Bases: `object`

This contains the Residential model and its associated functions. Once an object is instantiated, it can calculate new Load values for updated prices. It can also calculate estimated changes to the Load if one of the input variables is changed by a specified percent. The model will be created in a symbolic form to be easily manipulated, and then values can be filled in for calculations.

`baseYear` **= 0**

`complex_step_sensitivity` (*prices, change_var, percent*)

This estimates how much the output Load will change due to a change in one of the input variables. It can calculate these values for changes in price, price elasticity, income, income elasticity, or long term trend. The Load calculation requires input prices, so this function requires that as well for the base output Load. Then, an estimate for Load is calculated for the case where the named 'change_var' is changed by 'percent' %.

Parameters

prices : *dataframe or Pyomo Indexed Parameter*
Price values used to calculate the Load value

change_var : *string*

Name of variable of interest for sensitivity. This can be:
'income', 'i_elas', 'price', 'p_elas', 'trendGR'

percent : *float*
A value 0 - 100 for the percent that the variable of interest can change.

Returns

dataframe
Indexed values for the calculated Load at the given prices, the Load if the variable of interest is increased by 'percent'%, and the Load if the variable of interest is decreased by 'percent'%

`hr_map` **= Empty DataFrame Columns: [] Index: []**

`loads` **= {}**

`make_block` (*prices, pricesindex*)

Updates the value of 'Load' based on the new prices given. The new prices are fed into the equations from the residential model. The new calculated Loads are used to constrain 'Load' in pyomo blocks.

Parameters

prices : *pyo.Param*
Pyomo Parameter of newly updated prices

pricesindex : *pyo.Set*
Pyomo Set of indexes that matches the prices given

Returns

pyo.Block
Block containing constraints that set 'Load' variable equal to the updated load values

`prices` **= {}**

`sensitivity` (*prices, change_var, percent*)

This estimates how much the output Load will change due to a change in one of the input variables. It can calculate these values for changes in price, price elasticity, income, income elasticity, or long term trend. The Load calculation requires input prices, so this function requires that as well for the base output Load. Then, an estimate for Load is calculated for the case where the named 'change_var' is changed by 'percent' %.

Parameters

prices : *dataframe or Pyomo Indexed Parameter*
Price values used to calculate the Load value

change_var : *string*

Name of variable of interest for sensitivity. This can be:
'income', 'i_elas', 'price', 'p_elas', 'trendGR'

percent : *float*
A value 0 - 100 for the percent that the variable of interest can change.

Returns

dataframe
Indexed values for the calculated Load at the given prices, the Load if the variable of interest is increased by 'percent'%, and the Load if the variable of interest is decreased by 'percent'%

`update_load` (*p*)
Takes in Dual pyomo Parameters or dataframes to update Load values

Parameters

p : *pyo.Param*
Pyomo Parameter or dataframe of newly updated prices from Duals

Returns

pandas DataFrame
Load values indexed by region, year, and hour

`view_output_load` (*values: DataFrame, regions: list[int] = [1], years: list[int] = [2023]*)
This is used to display the updated Load values after calculation. It will create a graph for each region and year combination.

Parameters

values : *pd.DataFrame*
The Load values calculated in update_load

regions : *list[int], optional*
The regions to be displayed

years : *list[int], optional*
The years to be displayed

`view_sensitivity` (*values: DataFrame, regions: list[int] = [1], years: list[int] = [2023]*)
This is used by the sensitivity method to display graphs of the calculated values

Parameters

values : *pd.DataFrame*
indexed values for the Load, upper change, and lower change

regions : *list[int], optional*
regions to be graphed

years : *list[int], optional*
years to be graphed

`src.models.residential.scripts.residential.`**`run_residential`** (*settings: Config_settings*)
This runs the residential model in stand-alone mode. It can run update_load to calculate new Load values based on prices, or it can calculate the new Load value along with estimates for the Load if one of the input variables changes.

Parameters

settings : *Config_settings*
information given from run_config to set several values

S

A

B

C

D

E

F

G

H

I

L

M

O

P

Q

R

S

T

U

V

W

Y