Multiscale Solar Water Heating - Code Documentation

Release 2.0.0

Milica Grahovac
Katie Coughlin
Mohan Ganeshalingam
Hannes Gerhart
Robert Hosbach
Vagelis Vossos

Dec 02, 2020

CONTENTS

CHAPTER

ONE

PACKAGE INSTALLATION

Most users will be able to install the MSWH Python package by following the Setup and Installation and Simple Installation Using Conda section of the README.md file that is displayed at the landing page of the MSWH repository. Please use those instructions as the primary approach to MSWH package installation.

The set of instructions presented here is intended for technical users that are relatively new to virtual environments or Python in general, for users who encountered issues with the simple installation instructions available in the README.md file, or any other users looking for a reminder on some of the installation steps. These instructions also show to the users how to utilize an alternative Python package management system, venv.

Please make sure to install `pip` and, in case you are not using `venv`, `conda` as instructed in Setup and Installation on the readme file.

Here are the detailed steps to install the MSWH Python package:

1. Since the repo comes with database files, please download, install and see the documentation for git large file storage.
2. It is recommended to create a new `Python` environment in order to avoid interference with the system-wide Python installation, for example by using conda or venv. Depending on the approach you take, pick one of the commands below and run it in a terminal to create a new environment named, for instance, `mswh`.

   If you use `conda` from the repo clone folder run:

   ```
   conda create -n mswh python=3.8
   ```

   If you use `venv`, for example on `Linux`:

   ```
   python3.8 -m venv <path_to_env>/mswh
   ```

   With `<path_to_env>` as your selected folder path to store virtual environments.
3. Now the virtual environment needs to be activated, by running one of the following commands:

   When using `Anaconda` or `Miniconda`:

   ```
   conda activate mswh
   ```

   When using `venv`:

   ```
   source <path_to_env>/mswh/bin/activate
   ```

   After having activated the virtual environment, the name of it should appear before the prompt in the terminal.

   For deactivating use:

```
conda deactivate
```

4. To make use of example `Jupyter notebooks` one should have JupyterLab installed. To ensure the same Python kernel can be used in a `Jupyter notebook`, activate the virtual environment and run:

   ```
   python -m ipykernel install --user --name mswh
   ```

   Users with admin privileges can skip the `--user` flag.

   If you have any issues with plots not being displayed when running the example notebooks, please install the following:

   ```
   jupyter labextension install jupyterlab-plotly
   ```

5. Clone the repository with:

   ```
   git clone https://github.com/LBNL-ETA/MSWH.git
   ```

6. To install the necessary Python packages navigate to the `setup.py` directory and run:

```
pip install -e .
```

   The `-e` flag is only necessary if one would like changes to the source code be reflected immediately (without having to rerun the `setup.py` script with every change to the source code). If you just want to run the project application, you can omit the `-e` flag.

7. To use the plotting capabilities, also required when running tests, please install orca.

CHAPTER

TWO

MULTISCALE SOLAR WATER HEATING (MSWH)

2.1 Scope

The main purpose of the Multiscale Solar Water Heating (MSWH) software is to model energy use for individual and community scale solar water heating projects in California.

The package contains functional and unit tests and it is structured so that it can be extended with further technologies, applications, and locations.

2.2 Usage

The user provides a climate zone for a project, occupancy for each household, and whether any of the occupants stay at home during the day. The software can then load a set of example California specific hourly domestic hot water end-use load profiles from a database, size, and locate the systems. Next, the user can simulate the hourly system performance over a period of one representative year, visualize and explore the simulation results using time-series plots for temperature profiles, heat and power rates, or look at annual summaries. Similarly, the user can model individual household solar water heating projects and base case conventional gas tank water heater systems, such that the results can be compared between the individual, community-scale, and base case systems.

This functionality is readily available through a Jupyter notebook and a Django web framework graphical user interface (GUI), depending on what level of detail the user would like to access. Please see the README file on the MSWH repo for further usage and installation instructions.

System performance time series visualizations are available both in example notebooks and through the GUI, either spun off locally or using a web deployed version.

2.3 Features

This software package contains the following Python modules:

- Solar irradiation on a tilted surface.
- Simplified component models for:
  - Converters: solar collectors, electric resistance heater, gas burner, photovoltaic panels, heat pump.
  - Storage: solar thermal tank, heat pump thermal tank, conventional gas tank water heater.
  - Distribution: distribution and solar pump, piping losses.
- Preconfigured system simulation models for:
  - Base case gas tank water heaters.

  - Solar thermal water heaters (solar collector feeding a storage tank, with a tankless gas water heater backup in a new installation cases and a basecase gas tank water heater in a retrofit case).
  - Solar electric water heaters (grid supported photovoltaic panel powering a heat pump storage tank with an electric resistance backup).
- Database with component performance parameters, California specific weather data, and domestic hot water end-use load profiles.
- Django web framework to configure project, parametrize components and simulate from a web browser.

We also developed component sizing rules and size scaling rules to account for the household occupancy and project scale, respectively. The rules are readily available in the example notebooks and can easily be modified for exploratory purposes that we further describe in the Statement of Need section. For the sizing and scaling rules, we used the following data sources: expert knowledge, web-scraped data with the help of a tool described in [GMT17], sizing rules available in [CPU16], and certification databases such as [USDoEAESProgram18] and [CEC19].

2.4 Approach to Component and System Modeling and Simulation

In this section we briefly introduce the characteristics of the underlying models and simulation.

We performed an extensive literature rewiew prior to developing the models. Modelica buildings library by [WZNP14] exceeds the level of detail but proves too detailed and thus somewhat slow for our particular application. SAM tool ([BDF+14]) has a fitting level of detail, provides most of the system models that we needed but for our purposes proves not flexible enough in terms of modifying the system configuration, automating the size scaling, and embedding it into our custom life-cycle cost framework.

Namely, to capture a sufficient level of detail of the California demographics, such as variability in climate zones, household types, and household occupancy, we wanted to be able to simulate a few alternative water heating systems in each of the California sample households. Secondly, to get a more realistic picture of the effect of thermal storage and distribution system losses, we opted to perform a simulation with relatively short time-steps of one hour for a duration of one representative year. We were not able to identify an open source tool that is capable of firstly satisfying the simulation speed requirement combined with the necessary level of detail for our analysis and secondly providing the flexibility for us to customize various integral parts of the analysis such as automate the component and system size scaling, specify hot water load profiles and solar radiation for each household or group of households in the sample.

To satisfy our research need we thus opted to develop lightweight simulation models for all involved systems that would allow for around 120,000 simulation runs together with the component sizing and life-cycle cost analysis to be performed on a computer with a 12-core processor in about 8 hours. The users can expect a single solar water heater simulation model to run in less than one second (the developers were experiencing run times on the order of 0.2 seconds), providing an almost instantaneous experience for a user only seeking to design and investigate a single system.

We developed and implemented simplified fast performing energy balance based component models. We connected the component models into two preconfigured solar water heating systems, that are both provided with the MSWH software. Those models are:

- Solar thermal collector, hot water thermal storage tank, with a selection of backups: gas storage water heater or an instantaneous gas water heater.
- Photovoltaic panel, heat pump tank water heater, with an electric resistance water heater as backup.

We built a simple simulation solver that uses the explicit forward Euler method to solve the balance equations in each simulation time-step.

The component models were either developed from scratch or implemented in Python based on existing models identified in the literature. We implemented the following existing or new models:

- Solar irradiation on a tilted surface model is based on equations found in [DB13].
- Solar collector models and model parameters are based on [ASH13] and [SRC13].
- We converted the natural gas tank water heater model from [LDWL+98] into an hourly time-step model implementation.
- Photovoltaic model is based on a simplified model found in [WZNP14].
- Heat pump water heater tank is based on [SHC11].
- Solar thermal tank is a phenomenological model based on ideas very similar to the model developed for NREL's SAM software ([BDF+14]), as described in [DCBD14].
- Simplified performance data-based gas burner model was implemented to represent instantaneous gas water heater.
- Simple electric resistance model was implemented to represent instantaneous electric water heater.
- We developed a simplified data based solar and distribution pump model.
- To model the distribution piping network we developed a simplified model that is capable of accounting for thermal losses at stagnation and flows on-demand with correction factors available to help account for the relatively long time-step of one hour.

More details on the hot water demand model used in creating the database of sample hot water use load profiles, as well as extensive detail on the software's solar radiation, component and system models can be found in the project report by [CGG+21]. [Ger19] thesis provides additional details on the solar electric system model development.

Note that the weather data are currently mostly limited to California and can be extended to other climate zones. An example climate zone outside of California was added for Banja Luka, Bosnia and Herzegovina, through an additional example Jupyter notebook. The water consumption profiles can be highly location specific and their development for additional climate zones would require new research efforts. A quick approximation may be made with caution by scaling the California profiles to match the location-specific estimate of the average annual water use. This is possible as the shape of each daily profile can be assumed similar and sufficiently variable to allow for the study of transient and peak load effects at any location. The weather processor is TMY3 enabled and the user may populate the database with additional climates as needed.

The energy sources we consider are solar irradiation, gas, and electricity. The source energy is converted, if needed stored, and distributed to meet the end-use loads for each household.

Upon assembling the components into systems, we perform an annual simulation with hourly timesteps. We solve any differential equations for each time step using an explicit forward Euler method, a first order technique that provides a good approximation given the dynamics of the process observed and the level of detail required in our analysis.

We configure and size each MSWH thermal configuration so that it complies with the CSI-T (California Solar Initiative - Thermal) rebate program sizing requirements. The system model assumes appropriate flow and temperature controls and includes freeze and stagnation protection.

2.5 Future Applications - Statement of Need

When it comes to the future application of the MSWH software, we can envision four main groups of users:

- Researchers and policy developers.
- Solar water heating planners, designers, and contractors.
- Homeowners.
- Educators.

If the features of the existing MSWH software are sufficient for their application, the policy developers and researchers could utilize the existing MSWH software by embedding it into some larger analysis framework they construct such that it provides answers to their specific research questions. Should they require additional system configurations and even additional components, the existing framework should be expanded in line with the structure made available to the user in the MSWH software. When systems are added following the structure of the existing systems, the addition of such a new system to the GUI is made possible by using the flexible web framework.

Solar thermal water heating system planners, designers, and contractors may find it useful to have access to a freely available simulation tool, such as the MSWH software, that they can use to evaluate various system designs. The design parameters that such users can easily modify are household occupancies, climate zone, collector and tank sizes, component performance parameters such as insulation level of any thermal storage tanks, and types of solar collectors. The MSWH software relies on standard collector rating data readily available for most designs found on the market today. For each proposed design the MSWH software will output, among other results, the solar fraction and the backup energy use on an annual level, the two variables allowing for a quick cross-comparison for the proposed designs.

Similarly, homeowners considering transitioning to a solar water heating system may be interested in analyzing a hypothetical system before seeking further professional help. Or, some homeowners may simply be interested in learning about both solar water heating systems and system simulation in general. Another example use case would be to enable the occupants of households that:

- Are retrofitting an existing system due to an increase or decrease in occupancy, or
- Already possess one of the components and are looking to appropriately size the others

to simulate alternatives and compare the obtained energy consumption and solar fraction results for any alternative designs they like to define.

Lastly, simulation tools tend to be inaccessible to non-technical users, both in terms of usage and the chance for the user to understand the underlying codebase just by reading through it. The MSWH software provides a unique insight into what actually happens in a relatively simple mezzo-level simulation model due to the use of readable Python code, while the example notebooks and GUI allow for instant utilization of the models. These features make the code suitable also for educators.

2.6 Code Development and Code Contributions

We welcome code contributions. The development primarily takes place on the MSWH GitHub repository. Please refer to the contributing guidelines and README.md for further instructions, including those on running the unit tests.

CHAPTER

THREE

PYTHON CODE DOCUMENTATION

3.1 Subpackages

3.1.1 System and Component Models

mswh.system.components module

class mswh.system.components.**Converter**(*params=None*, *weather=None*, *sizes=1.0*, *log_level=10*)

Bases: `object`

Contains energy converter models, such as solar collectors, electric resistance heaters, gas burners, photovoltaic panels, and heat pumps. Depending on the intended usage, the models can be used to determine either a time period of component operation (for example an entire year), or a single timestep of component performance.

Parameters:

params: pd df Component performance parameters per project Default: None (default model parameters will get used)

weather: pd df Weather data timeseries with columns: amb. temp, solar irradiation. Number of rows equals the number of timesteps. Default: None (constant values will be set - use for a single timestep calculation, or if passing arguments directly to static methods)

sizes: pd df Component sizes per project. Default: 1. (see individual components for specifics)

log_level: None or python logger logging level, Default: logging.DEBUG This applies for a subset of the class functionality, mostly used to deprecate logger messages for certain calculations. For Example: log_level = logging.ERROR will only throw error messages and ignore INFO, DEBUG and WARNING.

Note:

If more than one of the same component is a part of the system, a separate instance of the converter should be created for each instance of the component.

Each component is also implemented as a static method that can be used outside of this framework.

Examples:

See `mswh.system.tests.test_components` module and `scripts/Project Level MSWH System Tool.ipynb` for examples on how to use the methods as stand alone and in a system model simulation.

electric_resistance(*Q_dem*)

Electric resistance heater model. Can be used both as an instantaneous electric WH and as an auxiliary heater within the thermal tank.

Parameters:

Q_dem: float or array like, [W] Heat demand

Returns:

res: dict

- self.r['q_del_bckp'] : float, array - delivered heat rate, [W]
- self.r['q_el_use'] : float, array - electricity use, [W]
- self.r['q_unmet'] : float, array - unmet demand heat rate, [W]

gas_burner(*Q_dem*)

Gas burner model. Used both as an instantaneous gas WH and as a gas backup for solar thermal.

Parameters:

Q_dem: float or array like, W Heat demand

Returns:

res: dict

- self.r['q_del_bckp'] : float, array - delivered heat rate, [W]
- self.r['q_gas_use'] : float, array - gas use heat rate, [W]
- self.r['q_unmet'] : float, array - unmet demand heat rate, [W]

Any further unit conversion should be performed using unit_converters.Utility class

heat_pump(*T_wet_bulb*, *T_tank*)

Returns the current heating performance and electricity usage in the current conditions depending on wet bulb temperature, average tank water temperature, and the rated heating performance.

Rated conditions are: wet bulb = 14 degC, tank = 48.9 degC

Parameters:

T_wet_bulb: real, array Inlet air wet bulb temperature [K]

T_tank: real, array Water temperature in the storage tank [K]

C1: real Coefficient 1, either for normalized COP or heating capacity curve [-]

C2: real Coefficient 2, either for normalized COP or heating capacity curve [1/degC]

C3: real Coefficient 3, either for normalized COP or heating capacity curve [1/degC2]

C4: real Coefficient 4, either for normalized COP or heating capacity curve [1/degC]

C5: real Coefficient 5, either for normalized COP or heating capacity curve [1/degC2]

C6: real Coefficient 6, either for normalized COP or heating capacity curve [1/degC2]

Returns:

performance: dict

- 'cop': current Coefficient Of Performance (COP), [-]
- 'heat_cap': current heating capacity of heat pump, [W]
- 'el_use': current electricity use of heat pump [W]

photovoltaic (*use_p_peak=True*, *inc_rad=None*)
Photovoltaic model

Parameters:

use_p_peak: boolean Boolean flag determining if peak power is used for sizing the pv panel (instead of area and efficiency)

Returns:

self.pv_power: dict of floats Generated power [W]

- 'ac' : AC
- 'dc' : DC

property size

solar_collector (*t_in*, *t_amb=None*, *inc_rad=None*)
Two commonly used empirical instantaneous collector efficiency models based on test data from standard test procedures (SRCC, ISO9806), found in J. A. Duffie and W. A. Beckman, Solar engineering of thermal processes, 3rd ed. Hoboken, N.J: Wiley, 2006., are:

- Cooper and Dunkle (CD model, eq 6.17.7)
- Hottel-Whillier-Bliss (HWB model, eq 6.16.1, 6.7.6)

Parameters:

t_in: float, array Collector inlet temperature (timeseries) [K]

t_amb: float, array Ambient temperature (timeseries) [K] Default: None (to use data extracted from the weather df)

inc_rad: float, array Incident radiation (timeseries) [W] Default: None (to use data extracted from the weather df)

Returns:

res: dict or floats or arrays

{'Q_gain' [Solar gains from the gross collector area, [W]] 'eff' : Efficiency of solar to heat conversion, [-]

property weather

class mswh.system.components.**Distribution** (*params=None*, *sizes=1.0*, *fluid_medium='water'*, *timestep=1.0*, *log_level=10*)

Bases: object

Describes performance of distribution system components.

Parameters:

sizes: pd df Pandas dataframe with component sizes, or 1.

fluid_medium: string Default: 'water'. No other options implemented

timestep: float, h Duration of a single timestep, in hours, defaults to 1.

log_level: None or python logger logging level, Default: logging.DEBUG This applies for a subset of the class functionality, mostly used to deprecate logger messages for certain calculations. For Example: log_level = logging.ERROR will only throw error messages and ignore INFO, DEBUG and WARNING.

Note:

Each component is also implemented as a static method that can be used outside of this framework.

Examples:

See `mswh.system.tests.test_components` module and for examples on how to use the methods.

pipe_losses(*T_in=333.15*, *T_amb=293.15*, *V_tap=0.05*, *max_V_tap=0.1514*)
Thermal losses from distribution pipes.

Parameters:

T_in: float, K Hot water temperature at distribution pipe inlet

T_amb: float, K Ambient temperature

V_tap: float, m3/h Timestep draw volume

max_V_tap: float, m3/h Maximum draw volume, m3/h (design variable)

Returns:

res: dict ['heat_rate']: Loss heat rate, W

pump(*on_array=array([1., 1., 1., ..., 1., 1., 1.])*, *role='solar'*)
Solar and distribution pump energy use. Assumes a fixed speed pump.

Parameters:

on_array: array Pump on/off status for the chosen number of discrete timesteps Default: np.ones(8760) - on for a year in hourly timesteps.

role: string 'solar' : primary (solar collector) loop 'distribution' : secondary (distribution) loop

Returns:

en_use: float or array like

property size

class `mswh.system.components.`**Storage**(*params=None*, *size=1.0*, *type='sol_tank'*, *timestep=1.0*, *log_level=10*)

Bases: `object`

Describes performance of storage components, such as solar thermal tank, heat pump thermal tank, conventional gas tank water heater.

Parameters:

params: pd df Component performance parameters per project Default: None. See tests and examples on how to structure this input.

weather: pd df Weather data timeseeries (amb. temp, solar irradiation) Default: None. See tests and examples on how to structure this input.

size: pd df or float, m3 Tank size. Default 1. See tests and examples on how to structure this input.

type: string Type of storage component. Options:

- 'sol_tank' - indirect tank WH with a coil to circulate fluid heated by a solar collector
- 'hp_tank' - tank with an inbuilt heat pump 'wham_tank' - conventional gas tank water heater model based on a WH model from the efficiency standards analysis
- 'gas_tank' - conventional gas tank water heater (currently not implemented)

log_level: None or python logger logging level, Default: logging.DEBUG This applies for a subset of the class functionality, mostly used to deprecate logger messages for certain calculations. For Example: log_level = logging.ERROR will only throw error messages and ignore INFO, DEBUG and WARNING.

timestep: float, h Duration of a single timestep, in hours, defaults to 1.

Note:

Create a new instance of the class for each storage component.

Examples:

See `mswh.system.tests.test_components` module and `scripts/MSWH System Tool.ipynb` for examples on how to use the methods as stand alone and in a system model simulation.

`electric_tank_wh`()

Currently not implemented.

`gas_tank_wh`(*V_draw*, *T_feed*, *T_amb=291.48*)

Gas storage water heater model (*_gas_tank_wh*) wrapper.

Parameters:

V_draw: float or array like, m3/h Hourly water draw for a single timestep of an entire analysis period

T_feed: float or array like, K Temperature of water heater inlet water for a single timestep of an entire analysis period

T_amb: float or array like, K Temperature of space surrounding water heater Default: 65 degF

Returns:

res: dict

- self.r['q_del'] : float, array - delivered heat rate, [W]
- self.r['q_dem'] : float, array - demand heat rate, [W]
- self.r['q_gas_use'] : float, array - gas use heat rate, [W]
- self.r['q_unmet'] : float, array - unmet demand, [w]
- self.r['q_dump'] : float, array - dumped heat, [W]

Note:

Assuming no electricity consumption in this version.

Make sure to size the tank according to the recommended sizing rules, since the WHAM model does not apply to tanks that are not appropriately sized.

`setup_electric`()

Currently not implemented.

`setup_thermal`(*medium='water'*, *split_tank=True*, *vol_fra_upper=0.5*, *h_vs_r=6.0*, *dT_param=2.0*, *T_max=344.15*, *T_draw_set=322.04*, *insul_thickness=0.085*, *spec_hea_cond=0.04*, *coil_eff=0.84*, *tank_re=0.76*, *dT_err_max=2.0*, *gas_heater_autosize=False*)

Sets thermal storage variables related to:

- loss calculation
- distribution of net gains/losses within two tank volumes (upper and lower)

Parameters:

medimum: string Storage medium (for thermal defaults to 'water')

split_tank: boolean If true, the tank is observed as two volumes, upper and lower tank volume. If false, the tank is observed as a single tank

vol_fra_upper: float Fraction of storage volume assigned to the upper tank volume (applies to 'thermal' only) If split_tank set to False, the value is ignored

dT_param: float, K Used as:

- Maximum temperature difference expected to occur between the upper and the lower tank volume while charging
- In-tank-coil approach

h_vs_r: float Regression parameter - tank height/diameter ratio (based on web scraped data), default: 6.

T_max: float, K Maximum allowed fluid temperature in the thermal storage tank, defaults to 344.15 K = 71 degC.

T_draw_set: float, K Draw temperature used in the load calculation, defaults to 120 degF = 322.04 K = 48.89 degC

insul_thickness: float, m Insulation thickness Default: .04 m (1-2 inch gas, 2-3 inch electric, based on DOE residential water heaters energy efficiency standard (ECS) analysis)

spec_hea_cond: float, W/mK Specific heat conductivity of the insulation Default: .04 W/mK (:from library:*ModelicaBuidlings*)

coil_eff: float Simplified efficiency of the coil heat exchanger Used in modeling of indirect coil-in-tank water heaters It excludes the approach temperature and represents the remaining heat transfer inefficiency

tank_re: float Recovery efficiency of a gas tank water heater. Used for the Storage.gas_tank_wh model

dT_err_max: float Allowed dT error below the minimum tank temperature due to finite timestep length approximation

gas_heater_autosize: boolean There is a gas heater in the tank and it will be autosized based on the tank volume

property size

tap (*V_draw_load*, *T_tank*, *T_feed*, *dT_loss=0.0*, *T_draw_min=None*)

Calculates the water draw volume and heat content drawn from the top of an infinitely large adiabatic tank given the hot water demand, tank temperature and the water main temperature.

It functions somewhat similarly to a thermostatic valve since it regulates the tap flow from the tank as follows:

- Limits above if the tank temperature is higher than the nominal draw temperature
- Tap flow equals V_draw_load for any tank temperature between T_draw_min and T_draw_nom
- Tap flow is zero if tank temperature is below T_draw_min and T_draw_min is provided

The results represent the theoretical limit for the draw. The tank model will check if the full amount can be delivered or only a part of the demand, due to the limited tank volume and thermal losses from the tank, and adjust the values.

Parameters:

V_draw_load: float, m3/h Volume of DHW drawn at the nominal end-use load temperature.

T_tank: float, K Tank node temperature from which the DHW is being tapped (usually the upper volume)

T_feed: float or array, K Temperature of water heater inlet water

dT_loss: float, K Distribution loss temperature difference

T_draw_min: float, K Minimal temperature that needs to be achieved in the tank in order to allow tapping.

Default: None - tapping is always enabled

Recommended usage - in colder climates where an outdoors tank may be cooler than the water main.

Returns:

draw: dict

- Draw volume: 'vol', m3/h
- Total demand heat rate: 'tot_dem', W
- Infinite volume delivered heat rate: 'heat_rate', W
- Infinite volume unmet heat rate: 'unmet_heat_rate', W

thermal_tank(*pre_T_amb=293.15*, *pre_T_feed=291.15*, *pre_T_upper=328.15*, *pre_T_lower=323.15*, *pre_V_tap=0.00757*, *pre_Q_in=400.0*, *max_V_tap=0.1514*)

Model of a thermal storage tank with:

- Coil heat exchanger for the solar gains
- DHW tap at the top of the tank
- Recharge tap at the bottom of the tank

The model can be instantiated as a:

- Solar thermal tank
- Heat pump tank

Parameters:

type: string

- 'solar' - solar tank (assumes that heated fluid from a solar collector is circulated through an in-tank-coil)
- 'hp' - heat pump tank (assumes an inbuilt heat pump as a main heat source)

The type will affect output labeling and heat transfer efficiency.

pre_T_amb: float, K Ambient temperature

pre_T_feed: float, K Temperature of the water that replenishes the tapped volume (e.g. water main temperature)

pre_T_upper: float, K Upper tank volume temperature

pre_T_lower: float, K Lower tank volume temperature

pre_Q_in: float, W Heat gain passed to in-tank coil from solar collector or from a heat pump, depending on the type

pre_V_tap: float, m3/h Volume of water tapped from the top of the tank

max_V_tap: float, m3/h Annual peak flow

Returns:

res: dict Single timestep input and output values for temperatures [K] and heat rates [W]:

```
>>> {net_gain_label : pre_Q_in_net,
self.r['q_loss_low'] : pre_Q_loss_lower,
self.r['q_loss_up'] : pre_Q_loss_upper,
# demand, delivered and unmet heat
# (between tap setpoint and water main)
self.r['q_dem'] : tap['net_dem'],
self.r['q_dem_tot'] : tap['tot_dem'],
self.r['q_del_tank'] : tank[self.r['q_del_tank']],
self.r['q_unmet_tank'] : np.round(
tank[self.r['q_unmet_tank']] + tap['unmet_heat_rate'], 2),
self.r['q_dump'] : tank[self.r['q_dump']],
self.r['q_ovrcool_tank'] : tank[self.r['q_ovrcool_tank']],
self.r['q_dem_balance'] : np.round(Q_dem_balance),
# average temperatures for tank volumes
self.r['t_tank_low'] : tank[self.r['t_tank_low']],
self.r['t_tank_up'] : tank[self.r['t_tank_up']],
self.r['dt_dist'] : dist['dt_dist'],
self.r['t_set'] : self.T_draw_set,
self.r['q_dist_loss'] : dist['heat_loss'],
self.r['flow_on_frac'] : dist['flow_on_frac']}
Temperatures in K, heat rates in W
```

thermal_tank_dynamics(*pre_T_amb*, *pre_T_upper*, *pre_T_lower*, *pre_Q_in*, *pre_Q_loss_upper*, *pre_Q_loss_lower*, *pre_T_feed*, *pre_Q_tap*)

Partial model of a thermal storage tank. Applies first order forward marching Euler method and updates the tank state for the current timestep based on the enthalpy balance and simplified assumptions about stratification. Thus, all input variables pertain to the previous timestep, while the outputs are solutions for the current timestep.

For example partial model application see thermal_tank method.

See inline comments for detailed explanation of the model.

Parameters:

pre_T_amb: float, K Ambient air temperature

pre_T_upper: float, K Upper tank volume temperature

pre_T_lower: float, K Lower tank volume temperature

It is recommended to set equal initial values for pre_T_upper and pre_T_lower

pre_Q_in: float, W Total heat gain (e.g. from a coil heat exchanger, a heating element, etc.)

pre_Q_loss_upper: float, W Heat loss from the upper tank volume

pre_T_lower: float, W Heat loss from the lower tank volume

pre_T_feed: float, K Temperature of the water that replenishes the tapped volume (e.g. water main temperature)

pre_Q_tap: float, W Heat loss that would occur if the tank volume at pre_T_upper was infinite

Returns:

res: dict of floats Represent averages in a single timestep. Average temperatures for tank volumes:

- self.r[self.r['t_tank_low']] : lower, K

- self.r['t_tank_up'] : upper, K

Heat rates:

- 'Q_net' : expected timestep net gain/loss based on inputs, W self.r['q_dump'] : dumped heat, W
- 'Q_draw' : delivered to load W
- 'Q_draw_unmet' : unmet load due to finite tank volume, W self.r['q_ovrcool_tank'] : error in balancing due to minimal tank temperature limit assumption in each timestep

Note: 'Q_draw' + 'Q_draw_unmet' = pre_Q_tap

volume_to_power(*tank_volume*)

Method to convert a gas water heater's volume input power based on a linear regression performed on the web scraped data.

Parameters:

tank_volume: float or int Water heater tank volume [m3]

Returns

tank_input_power: float Water heater input (rated) power [W]

mswh.system.models module

class mswh.system.models.**System**(*sys_params=None*, *backup_params=None*, *weather=None*, *sys_sizes=1.0*, *backup_sizes=1.0*, *loads=None*, *timestep=1.0*, *log_level=10*)

Bases: object

Project level system models:

- Assembles system configurations
- Performs timestep simulation
- Returns annual and timestep project and household level results, such as gas and electricity use, heat delivered, unmet demand and solar fraction.

Parameters:

sys_params: pd df Main system component performance parameters per project Default: None (default model parameters will get used)

backup_params: pd df Backup system performance parameters per project. It should contain a household ID column, otherwise columns identical to params.

sys_sizes: pd df Main system component sizes Default: 1. (see individual components for specifics)

backup_sizes: pd df Backup system component sizes, contains household id column Default: 1. (see individual components for specifics)

weather: pd df Weather data timeseries. Number of rows equals the number of timesteps. Can be generated using the Source.irradiation_and_water_main method

Example:

```
>>> sourceASource(read_from_input_dataframes = inputs)
```

Oakland climate zone in CEC weather data is '03':

```
>>> self.weather = source.irradiation_and_water_main('03', method=
↪'isotropic diffuse')
```

loads: pd df A dataframe with loads for all individual household served by the project level system. It should contain 3 columns: household id, occupancy and a column with a load array in m3 for each household.

Example:

```
>>> loads_com = pd.DataFrame(data = [[1, occ_indiv - 1., 0.8 * load_
↪array],[2, occ_indiv, 1. * load_array],[3, occ_indiv, 1.2 * load_
↪array],[4, occ_indiv + 1., 1.4 * load_array]], columns = [self.c['id
↪'], self.c['occ'], self.c['load_m3']])
```

timestep: float, h Duration of a single timestep, in hours Default: 1. h

log_level: None or python logger logging level, Default: logging.DEBUG This applies for a subset of the class functionality, mostly used to deprecate logger messages for certain calculations. For Example: log_level = logging.ERROR will only throw error messages and ignore INFO, DEBUG and WARNING.

Examples:

See `mswh.system.tests.test_components` module and `scripts/MSWH System Tool.ipynb` for examples on how to use the methods as stand alone and in a system model simulation.

conventional_gas_tank()

Basecase conventional gas tank water heater. Make sure to size the tank according to the recommended sizing rules, since the WHAM model does not apply to tanks that are not appropriately sized.

Returns:

ts_proj: dict of arrays, W Heat:

- self.r['q_del']: delivered
- self.r['gas_use']: gas consumed

simulate(*type='gas_tank_wh'*)

Runs a 8760. hourly simulation of the provided system type.

Parameters:

type: string

- 'gas_tank_wh'
- 'solar_thermal_retrofit' (gas tank backup at each household)
- 'solar_thermal_new' (gas tankless backup at each household)
- 'solar_electric'

Returns:

en_use: dict Total energy use for the analysis period:

- 'gas', Wh
- 'electricity', Wh

sys_res: list List containing detailed system level output. See dedicated methods for details

solar_electric(*backup='electric'*)

Connects the components of the solar electric system and enables simulation.

Parameters:

backup: string electric - instantaneous WHs (new installations)

Returns:

sys_en_use: dict System level energy use for the analysis period: 'electricity', Wh

sol_fra: dict Solar fraction. Keys: 'annual', 'monthly'

ts_res: pd df Colums populated with state variable timeseries, such as average timstep heat rates and temperatures

res: dict Summarizes ts_res. Any heat rates are summed, while the temperatures are averaged for the analysis period (usually one year)

el_use: dict Electricity use broken into end uses 'dist_pump' - distribution pump, if present

rel_err: float Balancing error due to limitations of finite timestep averaging. More precisely, due to selecting minimum tank temperature as the lower between the water main and the ambient.

solar_thermal(*backup='gas'*)

Connects the components of the solar thermal system and simulates it in discrete timesteps.

Parameters:

backup: string retrofit - pulls from the basecase for each household gas, electric - instantaneous WHs (new installations)

Returns:

self.cons_total: pd df Consumer level energy use [W], heat rates [W], average temperatures [K], and solar fraction for the analysis period.

proj_total: pd series Project level energy use [W], heat rates [W], average temperatures [K], and solar fraction for the analysis period.

sol_fra: dict Solar fraction. Keys: 'annual', 'monthly'

pump_el_use: dict Electricity use broken into end uses 'dist_pump' - distribution pump, if present 'sol_pump' - solar pump

ts_res: pd df Timestep project level results for all energy uses [W], heat rates [W], temperatures [K], and the load.

backup_ts_cons: dict of dicts Timestep household level results for energy uses [W], and heat rates [W].

rel_err: float Balancing error due to limitations of finite timestep averaging.

property weather

mswh.system.source_and_sink module

class `mswh.system.source_and_sink.`**`SourceAndSink`**(*input_dfs=None*, *random_state=123*, *log_level=10*)

Bases: `object`

Generates timeseries that are inputs to the simulation model and are known prior to the simulation, such as outdoor air temperature and end use load profiles.

Parameters:

input_dfs: a dict of pd dfs Dictionary of input dataframes as read in from the input db by the `Sql` class (see example in `test_source_and_sink.SourceAndSinkTests.setUp`)

random_state: numpy random state object or an integer numpy random state object : if there is a need to maintain the same random seed throughout the analysis.

integer : a new random state object gets instanteated at init

log_level: None or python logger logging level, Default: logging.DEBUG This applies for a subset of the class functionality, mostly used to deprecate logger messages for certain calculations. For Example: log_level = logging.ERROR will only throw error messages and ignore INFO, DEBUG and WARNING.

static demand_estimate(*occ*)

Estimates gal/day demand as provided in the CSI-Thermal Program Handbook, April 2016 for installations with a known occupancy

Parameters:

occ: float Number of individual household occupants

irradiation_and_water_main(*climate_zone*, *collector_tilt='latitude'*, *tilt_standard_deviation=None*, *collector_azimuth=0.0*, *azimuth_standard_deviation=None*, *location_ground_reflectance=0.16*, *solar_constant_Wm2=1367.0*, *method='isotropic diffuse'*, *weather_data_source='cec'*, *single_row_with_arrays=False*)

Calculates the hourly total incident radiation on a tilted surface for any climate zone in California. If weather data from the provided database are passed as *input_dfs*, the user can specify a single climate.

Two separate methods are available for use, with all equations (along with the equation numbers provided in comments) as provided in J. A. Duffie and W. A. Beckman, Solar engineering of thermal processes, 3rd ed. Hoboken, N.J: Wiley, 2006.

Parameters:

climate_zone: string String of two digits to indicate the CEC climate zone being analyzed ('01' to '16').

collector_azimuth: float, default: 0. The deviation of the projection on a horizontal plane of the normal to the collector surface from the local meridian, in degrees. Allowable values are between +/- 180 degrees (inclusive). 0 degrees corresponds to due south, east is negative, and west is positive. Default value is 0 degrees (due south).

azimuth_standard_deviation: float, default: 'None' Final collector azimuth is a value drawn using a normal distribution around the collector_azimuth value with a azimuth_standard_deviation standard deviation. If set to 'None' the final collector azimuth equals collector_azimuth

collector_tilt: float, default: 'latitude' The angle between the plane of the collector and the horizontal, in degrees. Allowable values are between 0 and 180 degrees (inclusive), and values greater than 90 degrees mean that the surface has a downward-facing component. If a default flag is left unchanged, the code will assign latitude value to the tilt as a good approximation of a design collector or PV tilt.

tilt_standard_deviation: float, default: 'None' Final collector tilt is a value drawn using a normal distribution around the collector_tilt value with a tilt_standard_deviation standard deviation. If set to 'None' the final collector tilt equals collector_tilt

location_ground_reflectance: float, default: 0.16 The degree of ground reflectance. Allowable values are 0-1 (inclusive), with 0 meaning no reflectance and 1 meaning very high reflectance. For reference, fresh snow has a high ground reflectance of ~ 0.7. Default value is 0.16, which is the annual average surface albedo averaged across the 16 CEC climate zones.

method: string, default: 'HDKR anisotropic sky' Calculation method to use for estimating the total irradiance on the tilted collector surface. See notes below. Default value is 'HDKR anisotropic sky.'

solar_constant_Wm2: float, default: 1367. Energy from the sun per unit time received on a unit area of surface perpendicular to the direction of propagation of the radiation at mean earth-sun distance outside the atmosphere. Default value is 1367 W/m^2.

weather_data_source: string, default: 'cec' The type of weather data being used to analyze the climate zone for solar insolation. Allowable values are 'cec' and 'tmy3.' Default value is 'cec.'

single_row_with_arrays [boolean] A flag to reformat the resulting dataframe in a row of data where each resulting 8760 is stored as an array

Returns:

data: pd df Weather data frame with appended columns: 'global_tilt_radiation_Wm2', 'water_main_t_F', 'water_main_t_C', 'dry_bulb_C', 'wet_bulb_C', 'Tilt', 'Azimuth']

Notes:

The user can select one of two methods to use for this calculation:

1) **'isotropic diffuse':** This model was derived by Liu and Jordan (1963). All diffuse radiation is assumed to be isotropic. It is the simpler and more conservative model, and it has been widely used.

2) **'HDKR anisotropic sky':** This model combined methods from Hay and Davies (1980), Klucher (1979), and Reindl, et al. (1990). Diffuse radiation in this model is represented in two parts: isotropic and circumsolar. The model also accounts for horizon brightening. This is also a simple model, but it has been found to be slightly more accurate (and less conservative) than the 'isotropic diffuse' model. For collectors tilted toward the equator, this model is suggested.

3.1.2 Tools

mswh.tools.plots module

class `mswh.tools.plots.`**Plot**(*title='', label_h='Time [h]', label_v='Component performance', data_headers=None, save_image=True, legend=True, outpath='', duration_curve=False, boxmode='group', notebook_mode=False, width=1200, height=800, fontsize=28, legend_x=0.4, legend_y=1.0, margin_l=200.0, margin_b=200.0*)

Bases: `object`

Creates and saves plots to visualize and correlate arrays, usually timeseries

Parameters:

title: str Plot title

data_headers: list A list of labels in the same order as the corresponding data If None the labels will be the df column labels, or integer indices if a list got provided

label_h: str Horizontal axis label

label_v: str Vertical axis label

legend: boolean Plot the legend or not

save_image: boolean If True saves the created image with either a given or default path and filename. Supported file types are 'png' and 'pdf', as specified in the filename.

duration_curve: boolean If True it sorts the columns (df or arrays) and plots the duration_curve, returns a duration_curve metric as a real

outpath: string or '' (for current directory) Path to save the png image of the plot

boxmean: True, False, 'sd', 'Only Mean'

notebook_mode: boolean Plot in the notebook if True

width: int Image width

height: int Image height

fontsize: int Axis label font size

Returns:

fig: plotly figure if self.interactive else True

box(*dfs*, *plot_cols=None*, *groupby_cols=None*, *df_cat=None*, *outfile='box.png'*, *boxmean=False*, *colors=['#3D9970', '#FF4136', '#FF851B']*, *title='Energy Use'*, *boxpoints='outliers'*)

Creates box plots for the chosen *plot_col* and can group plots by the *groupby_col*.

Parameters:

dfs: list of dfs

df_cat: list of str Indicator of the category carried by the dfs (E.g. the dfs differ by housing type)

plot_col: list of columns to plot, one from each df in dfs. If multiple dfs are passed, the values will be shown as groups on the plot

groupby_cols: list of cols to use as x axis, from each df. Use the same column if it has the same elements. Use None if x axis category not used

boxpoints: False, 'all', 'outliers', 'suspectedoutliers' See https://plot.ly/python/reference/#box

Returns:

fig: plotly figure if self.interactive else True

scatter(*data*, *outfile='scatter.png'*, *modes='lines+markers'*)

Creates a scatter plot

Parameters:

data: array/list, pd series, list of arrays/lists, pd df Provide a list or arrays/lists or a pandas dataframe. The variables should be ordered in pairs such that each odd variable in the list/first column in the df gets assigned to the horizontal axis, each even variable to the vertical axes. Each pair needs to have the same length, but pairs can be of a different length.

outfile: str Filename, include .png, .png .pdf

modes: str or list of str 'markers', 'lines', 'lines + markers' or a list of the above to assign to each plot (one string in a list for each pair of data)

Returns:

fig: plotly figure if self.interactive else True

series (*data*, *index_in_a_column=None*, *outfile='series.png'*, *modes='lines+markers'*)

Plots all series data against either the index or the first provided series. It can sort the data and plot the duration_curve.

Parameters:

data: array/list, pd series, list of arrays/lists, pd df Provide an array or a list if plotting a single variable. If plotting multiple variables provide a list of arrays or a pandas dataframe.

Horizontal axis corresponds to:

- if pd df: the index of the dataframe or the first columns of the dataframe
- if list or arrays/lists: a range of array length of the first array/list in the list

All arrays in the list need to have the same length.

index_in_a_column: boolean Horizontal axis labels If None, dataframe index is used, otherwise pass a column label for a column (it will not be considered as a series to plot)

outfile: str Filename, include .png, .png .pdf

modes: str or list of str 'markers', 'lines', 'lines+markers' or a list of the above to assign to each column of data, excluding the first column if index_in_a_column is not None

Returns:

fig: plotly figure if self.interactive else True

mswh.tools.unit_converters module

class mswh.tools.unit_converters.**UnitConv** (*x_in*, *scale_in=1.0*, *scale_out=1.0*)

Bases: `object`

Unit conversions using conversion parameters from ASHRAE Fundamentals 2017.

Parameters:

x_in: float, array Input value to be converted to a desired unit

scale_in: str or 1. Scale of the input value, options: 'k', 'kilo', 'mega', 'million', 'M', 'MM', 'giga', 'G', 'tera', 'T', 'peta', 'P', 'milli', 'micro'. Default: 1.

scale_out: str or 1. Scale of the input value, options: 'k', 'kilo', 'mega', 'million', 'M', 'MM', 'giga', 'G', 'tera', 'T', 'peta', 'P', 'milli', 'm', 'micro'. Default: 1.

Examples:

To convert temperature from degF to degC

```
>>> t_in_degC = UnitConv(t_in_degF).degF_degC(unit_in='degF')
```

To convert power in hp to kW:

```
>>> p_in_kW = UnitConv(p_in_hp, scale_out='kilo').hp_W(unit_in='hp')
```

To convert energy from GJ to MMBtu:

```
>>> e_MMBtu = UnitConv(e_GJ, scale_in='G', scale_out='MM').Btu_J(unit_in='J')
```

Btu_J(*unit_in='Btu'*)
Converts work / energy / heat content between Btu and joule

Parameters:

x: float, array Input value

unit_in: string, options: 'Btu', 'J' Unit of the input value

Returns:

x_out: float, array Output value

Wh_J(*unit_in='J'*)
Converts work / energy / heat content between watthour and joule

Parameters:

x: float, array Input value

unit_in: string, options: 'Wh', 'J' Unit of the input value

Returns:

x_out: float, array Output value

degC_K(*unit_in='degC'*)
Converts temperature between degree Celsius and Kelvin

Parameters:

unit_in: string, options: 'K', 'degC' Unit of the input value

Returns:

x_out: float, array Output value

degF_degC(*unit_in='degF'*)
Converts temperature between degree Fahrenheit and Celsius

Parameters:

unit_in: string, options: 'degF', 'degC' Unit of the input value

Returns:

x_out: float, array Output value

ft_m(*unit_in='ft'*)
Converts length between foot and meter

Parameters:

x: float, array Input value

unit_in: string, options: 'Wh', 'J' Unit of the input value

Returns:

x_out: float, array Output value

hp_W(*unit_in='hp'*)
Converts power between watt and horsepower

Parameters:

unit_in: string, options: 'hp', 'W' Unit of the input value

Returns:

x_out: float, array Output value

m3_gal (*unit_in='gal'*)
Converts volume between cubic meter and gallon

Parameters:

unit_in: string, options: 'm3', 'gal' Unit of the input value

Returns:

x_out: float, array Output value

m3perh_m3pers (*unit_in='m3perh'*)
Converts volume flow between cubic meter per hour and cubic meter per second

Parameters:

x: float, array Input value

unit_in: string, options: 'Wh', 'J' Unit of the input value

Returns:

x_out: float, array Output value

sqft_m2 (*unit_in='sqft'*)
Converts area between square foot and square meter

Parameters:

x: float, array Input value

unit_in: string, options: 'Wh', 'J' Unit of the input value

Returns:

x_out: float, array Output value

therm_J (*unit_in='therm'*)
Converts work / energy / heat content between therm and joule

Parameters:

x: float, array Input value

unit_in: string, options: 'therm', 'J' Unit of the input value

Returns:

x_out: float, array Output value

class mswh.tools.unit_converters.**Utility** (*quantity_in*)
Bases: object

Converts gas or electricity consumption into commonly used units.

Parameters:

quantity_in: float, array Quantity to be converted. E.g. gas use in kJ

gas (*unit_in='kJ'*, *unit_out='MMBtu'*)
Converts gas consumption.

Parameters:

unit_in: string Units of the input quantity that needs to be converted. Options: 'kWh', 'kJ'

unit_out: string Desired output unit Options: 'm3', 'cf', 'therm', 'MMBtu'

Returns:

gas_use: float Gas use in output units

3.1.3 Database Communication

mswh.comm.sql module

class mswh.comm.sql.**Sql**(*path_OR_dbconn*)

Bases: `object`

Performs python-sqlite db communication.

Parameters:

path_OR_dbconn: str or a database connection instance Full path to a database file or an already instantiated connection object

commit(*sql_command*, *close=False*)

Execute a custom sql command

Parameters:

sql_command: string sql_command to execute

Returns:

close: boolean, default=False If True, closes the connection to db

csv2table(*path_to_csv*, *table_name*, *column_label_row=0*, *converters=None*, *close=False*)

Use to update bulk price or performance data. If same named table exists, it gets replaced

Parameters:

path_to_csv: str Full path to the csv table

table_name: str sql table name of choice

column_label_row: int, default=0 Index of the row which gets converted into column labels

converters: dict, default=None According to pandas documentation: Dict of functions for converting values in columns. Keys can be integers or column labels.

close: boolean, default=False If True, closes the connection to db

pd2table(*df*, *table_name*, *close=False*)

Write a dataframe out to the database. If same named table exists, it gets replaced

Parameters:

table_name: str sql table name

close: boolean, default=False If True, closes the connection to db

table2pd(*table_name*, *column_label_row=0*)

Reads in a single sql table.

Parameters:

table_name: str sql table name

column_label_row: int, default=0 Index of the row which gets converted into column labels

Returns:

df: pandas dataframe Sql table read in as a pandas df.

`tables2dict`(*close=True*)

Reads all tables contained in a sql database and converts them to a pandas dataframe.

Parameters:

close: boolean, default=True If True, closes the connection to db

Returns:

data: dict of pandas dataframes Saves each of the sql tables as a pandas dataframe under a sql table name as a key

CHAPTER FOUR

ACKNOWLEDGEMENTS

This work was supported by the California Energy Commission, Public Interest Energy Research Program, under Contract No. PIR-16-022.

CHAPTER

FIVE

REFERENCES

CHAPTER

SIX

COPYRIGHT NOTICE

CHAPTER SEVEN

LICENSE AGREEMENT

BIBLIOGRAPHY

[ASH13] ASHRAE. Ashrae 93:2003 - Method Of Testing To Determine The Thermal Performance Of Solar Collectors. 2013.

[BDF+14] Nate Blair, Aron P Dobos, Janine Freeman, Ty Neises, Michael Wagner, Tom Ferguson, Paul Gilman, and Steven Janzou. System Advisor Model, SAM 2014.1.14: General Description. *National Renewable Energy Laboratory (NREL), Golden, CO*, pages 19, 2014.

[CEC19] CEC. Appliance Certification Database. 2019. URL: https://cacertappliances.energy.ca.gov/.

[CGG+21] Katie Coughlin, Milica Grahovac, Mohan Ganeshalingam, Robert Hosbach, and Vagelis Vossos. Costs and Benefits of Community versus Individual End-use Infrastructure for Solar Water Heating. Technical Report, California Energy Commission, 2021.

[CPU16] CPUC. California Solar Initiative - Thermal Program Handbook. Technical Report, California Energy Commission, 2016.

[DCBD14] Nicholas DiOrio, Craig Christensen, Jay Burch, and Aron Dobos. *Technical Manual for the SAM Solar Water Heating Model*. 2014. URL: https://sam.nrel.gov/solar-water-heating.html.

[DB13] John A Duffie and William A Beckman. *Solar Engineering of Thermal Processes*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2013. ISBN 978-0-470-87366-3.

[Ger19] Hannes Gerhart. Implementation of a Flexible Web Framework for Simulating Python System Models. 2019.

[GMT17] Brian F. Gerke, Michael A. McNeil, and Thomas Tu. The International Database of Efficient Appliances (IDEA): A new tool to support appliance energy-efficiency deployment. *Applied Energy*, 205:453–464, November 2017. URL: http://www.sciencedirect.com/science/article/pii/S0306261917309650, doi:10.1016/j.apenergy.2017.07.093.

[LDWL+98] James D. Lutz, Camilla Dunham Whitehead, Alex Lekov, David Winiarski, and Gregory Rosenquist. WHAM: A Simplified Energy Consumption Equation for Water Heaters. In *472246*, volume 1, 171–183. ACEEE, March 1998. URL: http://aceee.org/files/proceedings/1998/data/papers/0114.PDF.

[SHC11] B. Sparn, K. Hudon, and D. Christensen. Laboratory performance evaluation of residential integrated heat pump water heaters. *National Renewable Energy Lab. (NREL), Golden, CO*, 9 2011. doi:10.2172/1025650.

[SRC13] SRCC. SRCC Standard 100 - Minimum Standards For Solar Thermal Collectors. January 2013. URL: https://solar-rating.org/programs/og-100-program/.

[WZNP14] Michael Wetter, Wangda Zuo, Thierry S. Nouidui, and Xiufeng Pang. Modelica Buildings library. *Journal of Building Performance Simulation*, 7:253–270, July 2014. URL: http://simulationresearch.lbl.gov/modelica/, doi:10.1080/19401493.2013.765506.

[USDoEAESProgram18] U.S. Department of Energy - Appliance & Equipment Standards Program. Compliance Certification Management System. 2018. URL: https://www.regulations.doe.gov/ccms.

PYTHON MODULE INDEX

m