oemof-demand

Release 0.2.3a2

oemof developer group

Apr 11, 2026

CONTENTS

CHAPTER

ONE

OVERVIEW

docs	
tests	
package	

Creating heat and power demand profiles from annual values.

- Free software: MIT license

1.1 Installation

```
pip install oemof.demand
```

You can also install the in-development version with:

```
pip install https://github.com/oemof/oemof-demand/archive/dev.zip
```

1.2 Documentation

https://oemof-demand.readthedocs.io/

1.3 Development

To run all the tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER

TWO

INSTALLATION

At the command line:

```
pip install oemof.demand
```

CHAPTER

THREE

BDEW LOAD PROFILES

Using the oemof-demand you can create heat and electrical profiles by scaling the BDEW profiles to your desired annual demand. The BDEW profiles are the standard load profiles from BDEW.

3.1 Heat Profiles

3.1.1 Description

Heat profiles are created according to the approach described in the corresponding BDEW guideline.

The method was originally established in this PhD Thesis at TU Munich.

The approach for generating heat demand profiles is described in section 4.1 (Synthetic load profile approach).

$$Q_{day}(\theta) = KW \cdot h(\theta) \cdot F \cdot SF$$

KW: Kundenwert (customer value). Daily consumption of customer at $\approx 8°C$, depending on SLP type and Temperature timeseries.
h: h-Wert (h-value) , depending on SLP type and daily mean temperature.
F: Wochentagsfaktor (week day factor), depending on SLP type and day of the week.
T: Daily mean temperature 2 meters above the ground (simple mean or "geometric series", which means a weighted sum over the previous days).
SF: Stundenfaktor (hour factor)

The geometric series approach is meant to account for thermal inertia.

$$\theta = \frac{T_t + 0.5 \cdot T_{t-1} + 0.25 \cdot T_{t-2} + 0.125 \cdot T_{t-3}}{1 + 0.5 + 0.25 + 0.125}$$

Depending on the profile type, different coefficients A, B, C, D for the sigmoid function are used.

$$h(\theta) = \frac{A}{1 + (\frac{B}{\theta - \theta_0})^C} + D$$

$$\theta_0 = 40°C$$

Types of houses:

EFH: Einfamilienhaus (single family house)
MFH: Mehrfamilienhaus (multi family house)
GMK: Metall und Kfz (metal and automotive)
GHA: Einzel- und Großhandel (retail and wholesale)

GKO: Gebietskörperschaften, Kreditinstitute und Versicherungen (Local authorities, credit institutions and insurance companies)
GBD: sonstige betriebliche Dienstleistung (other operational services)
GGA: Gaststätten (restaurants)
GBH: Beherbergung (accommodation)
GWA: Wäschereien, chemische Reinigungen (laundries, dry cleaning)
GGB: Gartenbau (horticulture)
GBA: Backstube (bakery)
GPD: Papier und Druck (paper and printing)
GMF: haushaltsähnliche Gewerbebetriebe (household-like business enterprises)
GHD: Summenlastprofil Gewerbe/Handel/Dienstleistungen (Total load profile Business/Commerce/Services)

Building class:

The parameter `building_class` (German: Baualtersklasse) can assume values in the range 1-11.

3.1.2 Usage

```
from oemof.demand import bdew

...
```

3.2 Electrical Profiles

3.2.1 Description

The electrical profiles are the standard load profiles from BDEW. All profiles have a resolution of 15 minutes. They are based on measurements in the German electricity sector. There is a dynamic function (h0_dyn) for the houshold (h0) profile that better takes the seasonal variance into account [BDEW].

$$F_t = -3,92 \cdot 10^{-10} \cdot t^4 + 3,2 \cdot 10^{-7} \cdot t^3 \breve{} 7,02 \cdot 10^{-5} \cdot t^2 + 2,1 \cdot 10^{-3} \cdot t + 1,24$$

With *t* the day of the year as a decimal number.

The following profile types are available. Be aware that the types in Python code are strings in **lowercase**.

Table 1: German (original) [Wikipedia]

Typ	Beschreibung	Erläuterung
G0	Gewerbe allgemein	Gewogener Mittelwert der Profile G1-G6
G1	Gewerbe werktags 8–18 Uhr	z.B. Büros, Arztpraxen, Werkstätten, Verwaltungseinrichtungen
G2	Gewerbe mit starkem bis überwiegendem Verbrauch in den Abendstunden	z.B. Sportvereine, Fitnessstudios, Abendgaststätten
G3	Gewerbe durchlaufend	z.B. Kühlhäuser, Pumpen, Kläranlagen
G4	Laden/Friseur	
G5	Bäckerei mit Backstube	
G6	Wochenendbetrieb	z.B. Kinos
G7	Mobilfunksendestation	durchgängiges Bandlastprofil
L0	Landwirtschaftsbetriebe allgemein	Gewogener Mittelwert der Profile L1 und L2
L1	Landwirtschaftsbetriebe mit Milchwirtschaft/Nebenerwerbs-Tierzucht	
L2	Übrige Landwirtschaftsbetriebe	
H0/H0_dy	Haushalt/Haushalt dynamisiert	

Table 2: British English (translation)

type	description	explanation
G0	General trade/business/commerce	Weighted average of profiles G1-G6
G1	Business on weekdays 8 a.m. - 6 p.m.	e.g. offices, doctors' surgeries, workshops, administrative facilities
G2	Businesses with heavy to predominant consumption in the evening hours	e.g. sports clubs, fitness studios, evening restaurants
G3	Continuous business	e.g. cold stores, pumps, sewage treatment plants
G4	Shop/barber shop	
G5	Bakery with bakery	
G6	Weekend operation	e.g. cinemas
G7	Mobile phone transmitter station	continuous band load profile
L0	General farms	Weighted average of profiles L1 and L2
L1	Farms with dairy farming/part-time livestock farming	
L2	Other farms	
H0/H0_dy	Household/dynamic houshold	

Further information in German language is available at the BDEW.

3.2.2 Usage

```
from oemof.demand import bdew
e_slp = bdew.ElecSlp(year=2020)

# get all available types
print(e_slp.get_profiles().columns)

# get the "h0" and "g0" profile
profiles = e_slp.get_profiles("h0", "g0")
```

(continues on next page)

(continued from previous page)

```
# get scaled profiles
scaled_profiles = e_slp.get_scaled_profiles({"h0": 3000, "g0": 5000})

# get scaled profiles with power values instead of energy values
# a conversion_factor of 4 will convert Wh, kWh etc. to W, kW
e_slp.get_scaled_power_profiles({"h0": 3000, "g0": 5000}, conversion_factor=4)

# add holidays, holidays are treated as Sundays
holidays = {
    datetime.date(2010, 1, 1): "New year",
    datetime.date(2010, 10, 3): "Day of German Unity",
}
e_slp = bdew.ElecSlp(year=2010, holidays=holidays)

# holiday dictionaries can be created using workalendar
# https://github.com/workalendar/workalendar
```

CHAPTER

FOUR

VDI4655 LOAD PROFILES

4.1 Overview

The VDI 4655 module implements load profile generation for residential buildings according to the German engineering standard VDI 4655. Heat and power demand profiles are generated based on typical days and building characteristics.

Key Features:

- Generates heating, hot water and power demand profiles for residential buildings
- Supports both single-family (EFH) and multi-family houses (MFH)
- Considers weather data, based on the building location in Germany
- Accounts for seasonal variations and holidays
- Customizable temperature limits for season definitions
- Adjustable temporal resolution (e.g., hourly, 15-minute intervals)

4.2 Example Usage

Here's a basic example of how to use the VDI 4655 module:

```
from oemof.demand import vdi

# Define houses
houses = [
    {
        "name": "EFH_1",
        "house_type": "EFH",
        "N_Pers": 3,
        "N_WE": 1,
        "Q_Heiz_a": 6000,
        "Q_TWW_a": 1500,
        "W_a": 5250,
        "summer_temperature_limit": 15,
        "winter_temperature_limit": 5,
    }
]

# Create region
region = vdi.Region(
    2017,
```

(continues on next page)

(continued from previous page)

```
    climate=vdi.Climate().from_try_data(try_region=4),
    houses=houses,
    resample_rule="1h"
)

# Generate load curves
load_curves = region.get_load_curve_houses()
```

4.3 House Parameters

Required parameters for each house:

- `name`: Unique identifier for the house
- `house_type`: Either "EFH" (single-family) or "MFH" (multi-family)
- `N_Pers`: Number of persons, up to 12 (relevant for EFH)
- `N_WE`: Number of apartments, up to 40 (relevant for MFH)
- `Q_Heiz_a`: Annual heating demand in kWh
- `Q_TWW_a`: Annual hot water demand in kWh
- `W_a`: Annual electricity demand in kWh

Optional parameters:

- `summer_temperature_limit`: Temperature threshold for summer season (default: 15°C)
- `winter_temperature_limit`: Temperature threshold for winter season (default: 5°C)

4.4 Weather Data

The module uses German test reference year (TRY) weather data by 'Deutscher Wetterdienst' (DWD) for determining the daily temperature and cloud coverage. You can:

- Use the weather data from one of the 15 TRY regions by DWD from 2010
  - Specify a TRY region number (`try_region` parameter), or
  - Use geographical coordinates to determine the TRY region (requires geopandas)
- Provide your own weather file (`file_weather` parameter), adhering to the standard of the TRY weather data published in 2016 by DWD (available at https://kunden.dwd.de/obt/)

4.5 Further Reading

For more details about the VDI 4655 standard, refer to:

- VDI 4655: Reference load profiles of single-family and multi-family houses for the use of CHP systems
- May 2008 (ICS 91.140.01)
- Verein Deutscher Ingenieure e.V.

CHAPTER

FIVE

FURTHER PROFILES

We implemented further profiles (one until now) to represent further demand sectors which are not covered by the BDEW load profiles.

5.1 Industrial Electrical Profile

5.1.1 Description

The industrial electrical profile uses a step function synthesized using different scaling factors for weekdays, weekend days and holidays as well as day time and night time.

5.1.2 Usage

The industrial profile is explained in the example *electricity_demand_example.py* located in the examples directory of the repository.

```
import datetime
import oemof.demand.particular_profiles as profiles
import pandas as pd

holidays = {
    datetime.date(2018, 1, 1): "New year",
}
# Set up IndustrialLoadProfile
ilp = profiles.IndustrialLoadProfile(
    dt_index=pd.date_range("01-01-2018", "01-01-2019", freq="15min"),
    holidays=holidays
)
# Get step load profile with own scaling factors and definition of
# beginning of workday
ind_elec_demand = ilp.simple_profile(
    annual_demand=1e4,
    am=datetime.time(9, 0, 0),
    profile_factors={
        "week": {"day": 1.0, "night": 0.8},
        "weekend": {"day": 0.8, "night": 0.6},
        "holiday": {"day": 0.2, "night": 0.2},
    },
)
```

CHAPTER

SIX

REFERENCE

Implementation of the standard load profiles

class oemof.demand.bdew.elec_slp.**ElecSlp**(*year*, *seasons=None*, *holidays=None*)

Bases: object

Generate electrical standardized load profiles based on the BDEW method.

Parameters

- **year** (*integer*) – Year of the demand series.
- **Optional Parameters**
- **————————-**
- **seasons** (*dictionary*) – Describing the time ranges for summer, winter and transition periods. The seasons dictionary will update the existing one, so only changed keys have to be defined. Make sure not to create time gaps. The "h0_dyn" will not work with changed seasons, so you have to use your own smoothing curve to create a "h0_dyn" profile.
- **holidays** (*dictionary or list*) – The keys of the dictionary or the items of the list should be datetime objects of the days that are holidays.

all_load_profiles(*time_df*, *holidays=None*)

create_bdew_load_profiles(*dt_index*, *slp_types*, *holidays=None*)

Calculates the hourly electricity load profile in MWh/h of a region.

create_dynamic_h0_profile()

property date_time_index

get_profile(*ann_el_demand_per_sector*)

DEPRECATED: Use *get_scaled_power_profiles()* instead

Parameters
ann_el_demand_per_sector (*dictionary*) – Key: sector, value: annual value

Returns
pandas.DataFrame (*Table with all profiles*)

get_profiles(**args*)

Get all or the selected profiles. To select profiles you can pass
the name of the types as strings. The profiles are normalised to 1.

Try *print(get_profiles().columns* to get all valid types.

Returns

pandas.DataFrame (*Table with all or the selected profiles.*)

Examples

```
>>> from oemof.demand import bdew
>>> e_slp = bdew.ElecSlp(year=2020)
>>> ", ".join(sorted(e_slp.get_profiles().columns))
'g0, g1, g2, g3, g4, g5, g6, h0, h0_dyn, l0, l1, l2'
>>> e_slp.get_profiles("h0", "g0").head()
                           h0        g0
2020-01-01 00:00:00  0.000017  0.000016
2020-01-01 00:15:00  0.000015  0.000015
2020-01-01 00:30:00  0.000014  0.000015
2020-01-01 00:45:00  0.000012  0.000014
2020-01-01 01:00:00  0.000012  0.000013
```

```
>>> e_slp.get_profiles("h0", "g0").sum()
h0    1.0
g0    1.0
dtype: float64
```

get_scaled_power_profiles(*ann_el_demand_per_sector*, *conversion_factor=4*)

Get profiles scaled by there annual value. Each value represents the average power of an interval. Therefore, it is not possible to sum up the array. A conversion factor is used to calculate power units from energy units. By default the conversion factor is *4*. As the interval of each profile is 15 minutes a conversion factor of *4* will convert energy units like Wh, kWh, MWh etc. to power units like W, kW, MW etc..

Parameters

- **ann_el_demand_per_sector** (*dict*) – The annual demand in an energy unit for each type.
- **conversion_factor** (*float*) – Factor to convert the energy unit of the annual value to the power unit of each interval.

Returns

pandas.DataFrame (*Table with scaled profiles.*)

Examples

```
>>> from oemof.demand import bdew
>>> e_slp = bdew.ElecSlp(year=2020)
>>> e_slp.get_scaled_power_profiles({"h0": 3000, "g0": 5000}).head()
                           g0        h0
2020-01-01 00:00:00  0.320338  0.202627
2020-01-01 00:15:00  0.305866  0.182365
2020-01-01 00:30:00  0.291590  0.164500
2020-01-01 00:45:00  0.278682  0.149633
2020-01-01 01:00:00  0.268122  0.138602
>>> cf = 4
>>> spp = e_slp.get_scaled_power_profiles({"h0": 3000, "g0": 5000},
...                                       conversion_factor=cf)
>>> spp.sum()
g0    20000.0
h0    12000.0
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> spp.div(cf).sum()
g0    5000.0
h0    3000.0
dtype: float64
```

get_scaled_profiles(*ann_el_demand_per_sector*)

Get profiles scaled by there annual value.

Parameters

ann_el_demand_per_sector (*dict*) – The annual demand in an energy unit for each type.

Returns

pandas.DataFrame (*Table with scaled profiles.*)

Examples

```
>>> from oemof.demand import bdew
>>> e_slp = bdew.ElecSlp(year=2020)
>>> e_slp.get_scaled_profiles({"h0": 3000, "g0": 5000}).head()
                           g0        h0
2020-01-01 00:00:00  0.080084  0.050657
2020-01-01 00:15:00  0.076466  0.045591
2020-01-01 00:30:00  0.072897  0.041125
2020-01-01 00:45:00  0.069671  0.037408
2020-01-01 01:00:00  0.067030  0.034650
```

```
>>> e_slp.get_scaled_profiles({"h0": 3000, "g0": 5000}).sum()
g0    5000.0
h0    3000.0
dtype: float64
```

oemof.demand.bdew.elec_slp.**dynamisation_function**(*timeindex: DatetimeIndex*) → Series

Use the dynamisation function of the BDEW to smoothen the seasonal edges. Functions resolution is daily.

$$F_t = -3,92 \cdot 10^{-10} \cdot t^4 + 3,2 \cdot 10^{-7} \cdot t^{3\smile} 7,02 \cdot 10^{-5} \cdot t^2 + 2,1 \cdot 10^{-3} \cdot t + 1,24$$

With *t* the day of the year as a decimal number.

Implementation of the bdew heat load profiles

class oemof.demand.bdew.heat_building.**HeatBuilding**(*df_index*, ***kwargs*)

Bases: `object`

Parameters

year (*int*) – year for which the profile is created

Variables

- **datapath** (*string*) – path to the bdew basic data files (csv)
- **temperature** (*pandas.Series*) – Series containing hourly temperature data
- **annual_heat_demand** (*float*) – annual heat demand of building in kWh

- **building_class** (*int*) – class of building according to bdew classification: possible numbers for EFH and MFH are: 1 - 11. Possible numbers for non-residential buildings are: 0.
- **shlp_type** (*string*) – type of standardized heat load profile according to bdew possible types are: GMF, GPD, GHD, GWA, GGB, EFH, GKO, MFH, GBD, GBA, GMK, GBH, GGA, GHA
- **wind_class** (*int*) – wind classification for building location (0=not windy or 1=windy)
- **ww_incl** (*boolean*) – decider whether warm water load is included in the heat load profile
- **ww_only** (*boolean*) – if true, only warm water load is included in the heat load profile

get_bdew_profile()

Calculation of the hourly heat demand using the bdew-equations

get_normalized_bdew_profile()

Calculation of the normalized hourly heat demand

get_sf_values(*filename='shlp_hour_factors.csv'*)

Determine the h-values

Parameters
filename (*string*) – name of file where sigmoid factors are stored

get_sigmoid_parameters(*filename='shlp_sigmoid_factors.csv'*)

Retrieve the sigmoid parameters from csv-files

Parameters
filename (*string*) – name of file where sigmoid factors are stored

get_temperature_interval()

Appoints the corresponding temperature interval to each temperature in the temperature vector.

get_weekday_parameters(*filename='shlp_weekday_factors.csv'*)

Retrieve the weekday parameter from csv-file

Parameters
filename (*string*) – name of file where sigmoid factors are stored

weighted_temperature(*how='geometric_series'*)

A new temperature vector is generated containing a multi-day average temperature as needed in the load profile function.

Parameters
how (*string*) – string which type to return ("geometric_series" or "mean")

Notes

Equation for the mathematical series of the average temperature[1]:

$$T = \frac{T_D + 0.5 \cdot T_{D-1} + 0.25 \cdot T_{D-2} + 0.125 \cdot T_{D-3}}{1 + 0.5 + 0.25 + 0.125}$$

with T_D = Average temperature on the present day
T_{D-i} = Average temperature on the day - i

[1] BDEW, # noqa: E501 BDEW Documentation for heat profiles.

References

Implementation of industrial step load profiles.

class oemof.demand.particular_profiles.**IndustrialLoadProfile**(*dt_index*, *holidays=None*, *holiday_is_sunday=False*)

Bases: `object`

Generate an industrial heat or electricity load profile.

simple_profile(*annual_demand*, ***kwargs*)

Create industrial step load profile.

Parameters

annual_demand (*float*) – Total demand over the period given upon initialisation of IndustrialLoadProfile through parameter *dt_index*. This is actually only the annual demand, if an entire year was given.

Other Parameters

- **am** (*datetime.time*) – Defines the beginning of the workday. Times between *am* and *pm*, including the start and end time defined by *am* and *pm*, are assigned "day" factors from *profile_factors*. Other times are assigned "night" factors. Default: 7 a.m.
- **pm** (*datetime.time*) – Defines the end of the workday. Times between *am* and *pm*, including the start and end time defined by *am* and *pm*, are assigned "day" factors from *profile_factors*. Other times are assigned "night" factors. Default: 11:30 p.m.
- **week** (*list(int)*) – List of weekdays, where 1 corresponds to Monday, 2 to Tuesday, etc. Weekdays are assigned "week" factors from *profile_factors*. Default: [1, 2, 3, 4, 5].
- **weekend** (*list(int)*) – List of weekend days, where 1 corresponds to Monday, 2 to Tuesday, etc. Weekend days are assigned "weekend" factors from *profile_factors*. Default: [6, 7].
- **holiday** (*list(int)*) – List of holiday days. Holidays given upon initialisation of the IndustrialLoadProfile object are tagged with weekday 0 if *holiday_is_sunday* is set to False, wherefore the default for this parameter is [0]. Holidays are assigned "holiday" factors from *profile_factors*. Default: [0].
- **profile_factors** (*dict*) – Dictionary with load profile scaling factors for night and day of weekdays, weekend days and holidays. The dictionary must have the same form as the dictionary given as the default value. Default:

```
{
    "week": {"day": 0.8, "night": 0.6},
    "weekend": {"day": 0.9, "night": 0.7},
    "holiday": {"day": 0.9, "night": 0.7},
}
```

Returns

pd.Series – Series with demand per time step (unit depends on the unit the *annual_demand* was provided with). Index is a DatetimeIndex containing all time steps the IndustrialLoadProfile was initialised with.

The region module combines profiles from VDI 4655 for heat and power demand of houses within one region.

This is an implementation of the calculation of load profiles defined in the German VDI 4655.

VDI 4655

Reference load profiles of single-family and multi-family houses for the use of CHP systems

May 2008 (ICS 91.140.10)

Verein Deutscher Ingenieure e.V.

VDI Standards Department

VDI-Platz 1, 40468 Duesseldorf, Germany

Reproduced with the kind permission of the Verein Deutscher Ingenieure e.V.

Notes

This script creates full year energy demand time series of domestic buildings for use in simulations. This is achieved by an implementation of the VDI 4655, which gives sample energy demands for a number of typical days ('Typtage'). The energy demand contains heating, hot water and electricity.

For a given year, the typical days can be matched to the actual calendar days, based on the following conditions: - Season: summer, winter or transition - Day: weekday or sunday (Or holiday, which counts as sunday) - Cloud coverage: cloudy or not cloudy - House type: single-family houses or multi-family houses (EFH or MFH)

class oemof.demand.vdi.regions.**Climate**(*temperature=None*, *cloud_coverage=None*, *energy_factors=None*)

Bases: `object`

Climate object for VDI time series.

Parameters

- **temperature** (*iterable of numbers*) – The ambient temperature in the area as daily mean values. The number of values must equal 365 or 366 for a leap year. Use the TRY data if no temperature data is available.
- **cloud_coverage** (*iterable of numbers*) – The cloud coverage in the area as daily mean values. The number of values must equal 365 or 366 for a leap year.

check_attributes()

from_try_data(*try_region*, *hoy=8760*)

class oemof.demand.vdi.regions.**Region**(*year*, *climate*, *seasons=None*, *holidays=None*, *houses=None*, *resample_rule=None*, *zero_summer_heat_demand=False*)

Bases: `object`

Define region-dependent boundary conditions for the load profiles.

After adding houses to the region, the load profiles for each house can be generated.

Parameters

- **year** (*int*) – Year of the profile.
- **seasons** (*dict (optional)*) – The times of the seasons if fixed seasons are used. In the VDI norm seasons are defined by the daily average temperature: Winter below 5 degree Celsius, Spring between 5 and 15 degree Celsius and summer above 15 degree Celsius.
- **holidays** (*dict or list (optional)*) – In case of a dictionary the keys are datetime objects and the values are strings with the name of the holiday. Otherwise a list of datetime objects should be passed.
- **houses** (*list (optional)*) – A list of dictionaries in which each house is defined the dictionary need to have the following keys.
- **resample_rule** (*str (optional)*) – Time interval to resample the profile e.g. 1h (1 hour) or 15min. The value will be passed to the pandas resample method.

- **zero_summer_heat_demand** (*bool (optional)*) – Set heat demand on all summer days to zero. Per default, multi-family houses have a small heat demand even in summer. (This is not part of VDI 4655)

`add_houses`(*houses*)

Add houses to the region object.

Parameters

houses (*list*) – A list of dictionaries that describes the houses.

Required parameters for each house:

- `name`: Unique identifier for the house
- `house_type`: Either "EFH" (single-family) or "MFH" (multi-family)
- `N_Pers`: Number of persons, up to 12 (relevant for EFH)
- `N_WE`: Number of apartments, up to 40 (relevant for MFH)
- `Q_Heiz_a`: Annual heating demand in kWh
- `Q_TWW_a`: Annual hot water demand in kWh
- `W_a`: Annual electricity demand in kWh

Optional:

- `summer_temperature_limit`: Temperature threshold for summer season (default: 15°C)
- `winter_temperature_limit`: Temperature threshold for winter season (default: 5°C)

`get_daily_energy_demand_houses`(*tl*)

Determine the houses' energy demand values for each 'typtag'.

Note

"The factors `F_el_TT` and `F_TWW_TT` are negative in some cases as they represent a variation from a one-year average. The values for the daily demand for electrical energy, `W_TT`, and DHW energy, `Q_TWW_TT`, usually remain positive. It is only in individual cases that the calculation for the typical-day category `SWX` can yield a negative value of the DHW demand. In that case, assume `F_TWW_SWX` = 0." (VDI 4655, page 16)

This occurs when `N_Pers` or `N_WE` are larger than their allowed maximum of 12 persons (for single-family houses) and 40 apartments (for multi-family houses).

`get_load_curve_houses`()

Generate time series of energy demand values for all houses.

This method calculates the energy demands heating, hot water, and electricity for each house in the region based on the VDI 4655 typical day profiles. The calculation uses daily energy demands and typical load profiles, which are combined to create a full year time series for each house and energy type.

The resulting time series are normalized to match the annual energy demands specified in the house parameters (Q_Heiz_a, Q_TWW_a, W_a).

Returns

pandas.DataFrame – MultiIndex DataFrame with the following structure:

- Index: DatetimeIndex with the time steps
- Columns: MultiIndex with levels

  - name: House identifier
  - house_type: Either “EFH” or “MFH”
  - energy: Energy type (“Q_Heiz_TT”, “Q_TWW_TT”, or “W_TT”)
- Values: Energy demand in kWh per time step

Read DWD TRY (test reference year) data.

`oemof.demand.vdi.dwd_try.`**`find_try_region`**(*longitude*, *latitude*)

Find the DWD TRY region by coordinates.

> **Note**
>
> - Latitude and longitude must be provided in the coordinate reference system `EPSG:4326`.
> - The packages geopandas and shapely need to be installed to use this function.

Parameters

- **longitude** (*float*)
- **latitude** (*float*)

Returns

DWD TRY region number (*int*)

Raises

`ImportError` – If geopandas or shapely are not installed

`oemof.demand.vdi.dwd_try.`**`read_dwd_weather_file`**(*weather_file_path*)

Read and parse DWD test reference year (TRY) weather data files.

This function reads TRY weather data files published by the German Weather Service (Deutscher Wetterdienst, DWD) and extracts temperature and cloud cover data.

The 2016 DWD weather files can be obtained from: https://kunden.dwd.de/obt/ (registration required)

Parameters

weather_file_path (*str, optional*) – Path to a TRY weather file. The file must follow the DWD format from 2010 or 2016. If None, a default file for the given try_region will be used.

Returns

pandas.DataFrame – DataFrame with hourly weather data

Raises

- **`TypeError`** – If the weather file does not follow the expected DWD format
- **`FileNotFoundError`** – If the weather file cannot be found

CHAPTER

SEVEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

7.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.2 Documentation improvements

oemof could always use more documentation, whether as part of the official oemof docs, in docstrings, or even on the web in blog posts, articles, and such.

7.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/oemof/oemof-demand/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

7.4 Development

To set up *oemof-demand* for local development:

1. Fork oemof-demand (look for the "Fork" button).
2. Clone your fork locally:

   ```
   git clone git@github.com:YOURGITHUBNAME/oemof-demand.git
   ```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

7.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

7.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

CHAPTER

EIGHT

AUTHORS

(alphabetic order)

- Amedeo Ceruti
- Benjamin Singh
- Birgit Schachler
- Caroline Möller
- Florian Maurer
- Francesco Witte
- Guido Plessmann
- Hendrik Huyskens
- Jann Launer
- Patrik Schönfeldt
- Pyosch
- Steffen Wehkamp
- Stephen Bosch
- Uwe Krien

CHAPTER

NINE

CHANGELOG

9.1 v0.2.3 (YYYY-MM-DD)

9.1.1 New features

9.1.2 Bug fixes

- Implement default temperature limits for VDI 4655 calculation to make `summer_temperature_limit` and `winter_temperature_limit` optional
- Fix leap year handling in VDI 4655 profiles

9.1.3 Other changes

- Documentation improvements.
- First version to be named oemof.demand.

9.2 v0.2.2 (2025-04-09)

- Added Electricity standard load profiles as released by the BDEW in 2025.
- Added VDI profile based on LPagg

9.3 v0.2.1 (2024-08-06)

9.3.1 New features

- Allow to have holidays in industrial profile

9.3.2 Bug fixes

- simple_profile from IndustrialLoadProfile behaved differently in version 0.2.0 compared to 0.1.9 due to pandas masking functions. We are now back to the old (design) values.

9.4 v0.2.0 (2024-06-27)

9.4.1 Bug fixes

- **Raise error for non supported shlp_type**
  in non-commercial buildings

9.4.2 Other changes

- Adhere to packaging standards
- Raise errors in sigmoid parameter queries

9.5 v0.1.9 (2023-03-18)

- Calculation of BDEW profiles was improved

9.6 v0.1.8 (2021-01-27)

9.6.1 Bug fixes

- FutureWarning for "dyn_function_h0" was raised instead of printed

9.7 v0.1.7 (2021-01-27)

9.7.1 New features

- Add dynamic h0 profile calculation (The implementation is not optimised for performance. Thus, it is not used by default.)

9.7.2 Bug fixes

- Fix improper use of pandas.dataframe.merge (oemof-demand will now work with pandas>=1.2)

9.7.3 Other changes

- Update deprecated pd.datetime to datetime.datetime
- Add (integration) tests and coverage as CI
- Split BDEW profile generation into submodules

9.8 v0.1.6 (2019-01-30)

9.8.1 General

- Update requirements
- Fix typos

9.9 v0.1.5 (2018-09-05)

9.9.1 New features

- Add function *get_normalized_bdew_profile(self)* to get a normalised profile. You could also use an annual_demand of one to get the same results.

9.9.2 Bug fixes

- Fix y-label of the heat example plot.

9.9.3 Other changes

- Make matplotlib optional in examples.

9.10 v0.1.4 (2018-05-30)

9.10.1 Code

- fix temperature bug
- fix Code style

9.10.2 Documentation

- Documentation improvements.

9.11 v0.1.1 (2016-11-30)

9.11.1 New features

- Examples callable by command-line script

9.11.2 Bug fixes

- Path specs when installed via pip

9.11.3 Other changes

- Fix versioning

9.12 0.1.0 (2016-10-04)

9.12.1 New features

- Implementation of BDEW synthetic load profiles
- Synthetic load profiles for heating sector
- Self-made industry demand profile similar to BDEW profiles

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

O

R

S

W