# DOOM Manual

## Prerequisites

* Python 3 (https://www.python.org/downloads/)
  + Pyomo
  + Numpy
  + Matplotlib
  + Pandas
* CoolProp (Achtung, hier muss man eventuell auf die richtige Python Version achtgeben!)
* Solver: LP/MILP, current setting: CPLEX
* Git
* IDE, e.g., PyCharm (optional)

To install python packages, use pip or conda.

Git pull the DOOM from GitLab (currently available: Koller-intern, EDCSproof, Sinfonies).

## File structure

The DOOM is modeled to be implementable as submodule for various projects. The integration of submodules requires some additional git commands. For more information on submodules please see [7.11 Git Tools – Submodules](https://git-scm.com/book/en/v2/Git-Tools-Submodules).

The git repository contains the modules “classes” and “auxiliary”. In the module “classes” all components are defined. The module “auxiliary” contains functions, which are used frequently and are not directly assigned to classes/components. In the folder “cases” you can find an “example\_case”, which is executable and shows how the setup of a use case works.

## Execution

Integrate the DOOM repository as submodule in your project or get a local DOOM repository. Optimally, a use case is generated outside of the repository. In this use case, the “classes” and “auxiliary” modules are imported, e.g. as “dc” and “da” (doom classes/doom auxiliary). Note, that you need to add the path, where the doom package is found, to the sys.path, so that the modules can be found.

E.g. add parent directory to sys.path:

Import os

import sys

PACKAGE\_PARENT = '..'

SCRIPT\_DIR = os.path.dirname(os.path.realpath(os.path.join(os.getcwd(), os.path.expanduser(\_\_file\_\_))))

sys.path.append(os.path.normpath(os.path.join(SCRIPT\_DIR, PACKAGE\_PARENT)))

If the doom package is found within the sys.path, then it should be importable as follows:

import doom.classes as dc

import doom.auxiliary as da

A use case always consists of one instance of the “System” class and any number of instances of the other unit classes. After the unit definition and addition to the system the nodes need to be added. Nodes are basically any balances, e.g., energy or mass balances. Note, that they can be modeled as equality but also as inequality constraints, if needed.

To run the program, just execute your use case script (should work just like the “example\_case”).

## General Description

The design and operation model (DOOM) may calculate the optimal design (unit size and occurrence) and operation of units within an energy supply system. The model considers discrete time steps for one or multiple periods (e.g. used for representative periods in order to calculate the optimal solution for a whole year).

The underlying optimization problem is an extended unit commitment problem with regard to the additional optimization of the unit sizes. For the complete UC formulation, binary variables (commimtment varialbes) are needed for each time step indicating the on/off status of a unit. Furthermore, the existence of a unit may as well be modeled with an additional binary variable, e.g. for fixed investment costs or for a certain minimum size. However, binary variables drastically increase the computational effort of the problem. Thus, depending on the input parameters of the units, binary variables are only used if necessary. For example, if a unit is modeled with a minimum part load, then for this unit the commitment variables are implemented. Therefore, it is possible to set up a use case without binary variables as well, which may be necessary due to the problem size.

### Total System Objective

The DOOM calculates the objective for a period times the weight of that period. This way it is possible, to optimize for any chosen time horizon, e.g., a whole year (TAC) for design optimization, but as well a single week for pure operation optimization (some objectives do not scale with the weight of the periods, e.g., the power based costs of the supply class; these costs have to be adapted according to the chosen time horizon).

The objective is built by the objectives of each unit. Each unit may have any objectives defined in the “.obj” dictionary of that unit. However, only the objective “total” will be added to the total system objective by the method “build\_model” of the “System” class and be considered by the solver. This way, unit specific objectives of interest for post analysis could be created. The system has two objectives, “total” and “total\_real”. In the objective “total” the costs of the slack variables of the nodes are included as well as some artificial small costs on all variables, which shall reduce oscillating solutions due to equivalent solutions, which are introduced in the “build\_model” method of the “System” class. The objective “total\_real” only has the real system costs included and is used for the system assessment.

### Use Case Structure

An example for a use case is provided in the folder “cases” as “example\_case.py”. The general structure is as follows:

* Import necessary modules
* Import data (e.g. time series)
* Define system parameters
* Generate system
* Define unit parameters, generate units by the “System” method “add\_unit”.
* Define node parameters, generate nodes by the “System” method “add\_nodes”.
* Complete the pyomo model by executing the “System” method “build\_model”.
* Add any custom variables, constraints and objectives, which are not implemented in the DOOM, if necessary.
* Solve the model via the “System” method “solve\_model”. Note, that the solved model is saved as pickle file.
* Do post analysis, e.g. generate plots.

## Modules

### Classes

#### General Information

The system and units are defined via dictionaries with the necessary input parameter.

There is a general structure for a system and units. A system has the instance attributes “param”, “unit”, “node”, “model”, “var”, “port”, “con” and “obj”. “var”, “port” and “con” are currently not used. The units have the instance attributes “var”, “port”, “con”, “obj” and “param”.

To define, whether units have binary variables, the parameters “u\_active”, “v\_w\_active” and “i\_active” are used. These paremeters are automatically set by the input parameters for a unit. If “u\_active” is “True”, then the commitment variable “u” is implemented for this unit (indexed variable with the size of the planning horizon). If “v\_w\_active” is “True”, then the startup and shut down variables “v” and “w” are implemented (indexed variable with the size of the planning horizon; note, these variables are not modeled as binary variables, since they are forces to be binary by the used constraints and the integrality/binary constraint of “u”). If “i\_active” is “True”, then the investment binary variable “i” is implemented (scaler variable). Many units use similar constraints or objectives; therefore, these are partly outsourced to the “auxiliary” module, e.g. the objective for investment costs.

The variables are separated in two types: “seq” for sequences/time series and “scalar”. The “scalar” variables are single variables. The “seq” variables are used to define the time dependent operation of a unit and are indexed variables with one dimension of the length of the planning horizon of one period and another dimension with the sizes of the number of periods (i.e. each period has its own operation variables). Furthermore, they may be additional dimensions, e.g., for heat pumps there is another dimension for the mass flow variable for the different inlets.

#### System

Every use case needs to be defined via one instance of the “System” class. For initialization, a dictionary with parameters needs to be provided, see “example\_case”. This class creates a pyomo ConcreteModel. Furthermore, there are methods to add units (“add\_unit”) and nodes (“add\_note”) as well as to add constraints and objective to the ConcreteModel (“build\_model”). Note, that only the objective “total” of a unit is added to the total system objective.

#### Unit

This class is the parent class of all units. It defines the basic structure of all units.

#### GasBoiler & ElectrodeBoiler

This unit represents a Fuel-to-Heat or Power-to-Heat unit. Note, that GasBoiler and ElectrodeBoiler classes are identical except for the naming of the variable and port “f” or “p”. The heat flow “q” and fuel/power consumption “f/p” is modeled in MW. The mass flow “m” is modeled in kg/s. The sizing variable ‘cap’ is a scalar, which defines the maximum heat flow “q”.

#### Cooler

The cooler class works just like the ElectrodeBoiler class, except that inlet and outlet are switch, i.e. the inlet is hotter that the outlet.

#### CHP

This unit represents a Fuel-to-Heat-and-Power unit. The unit may operate between 0-100% (or some minimal part load to 100%), which represents the heat generation “q”. The power generation “p” and fuel consumption “f” are linearly dependent on the heat generation. The class is capable of modeling varying efficiencies and power-to-heat ratios for part load to full load. Additionally, the mass flow is modeled as “m”. The variable “cap” represents the maximum heat generation and thus the size of the unit.

#### HeatPump

This class represents a heat pump. The heat flow at the sink and the source, as well as the electricity consumption are modeled. Furthermore, the inlet and outlet mass flows at the source and sink are modeled. There may be multiple inlets for sink and source, representing inlets from multiple temperature levels. However, at the outlet only one temperature level can be modeled (this is needed in order to model the COP linearly). The unit may operate between 0-100% (or some minimal part load to 100%), which represents the heat generation “q\_sink”. The COP may vary from part load to full load. It is calculated by the source inlet and outlet temperatures “T\_sink\_out/T\_source\_out”, the temperature differences “delta\_T\_sink/delta\_T\_source” at the heat exchangers and the efficiency of the compression “eta\_comp”. The temperature differences and the efficiency of the compression for minimal part load and full load are defined as input parameter. The variable “cap” represents the maximum heat generation and thus the size of the unit.

#### Photovoltaic

This class represents a photovoltaic system. It has the variables “p” for the power generation and “cap” for the size in m². As input the time series of the solar irradiation for all regarded periods have to be supplied. The power is calculated by the solar irradiation, the size/area and the efficiency “eta”. Note, that the power generation may be curtailed.

#### SolarThermal

This class represents a solar thermal system. This is similar to the class “Photovoltaic”, but instead of the power generation “p” there is the heat generation “q”. Furthermore, the mass flow is modeled. Optionally, there may be multiple inlets and outlets with different temperature levels. The temperature levels for inlets and outlets need to be defined. However, the distribution of the outlet mass flows needs to be predefined via the dictionary “m\_out\_dist” in the input parameter dictionary, e.g., in dependence of the amplitude of the solar irradiation, see “example\_case”.

#### HeatRecovery

This class represents a heat recovery system. The heat flow recovered “q” and the mass flow “m” are modeled as well as the maximum recovered heat flow “cap”, which is represents the unit size. The temperature levels for inlet and outlet need to be defined. A time series/sequence needs to be given, which is used as source of the heat recovery, e.g., a steam demand. The parameter “max\_recovery” defines the percentage of the recoverable heat. E.g., a value of 0.3 means that 30% of the given heat demand can be recovered by the heat recovery system.

#### Storage

This class represents an ideal generic storage model typically used in MILP models. The variables “c” and “d” define the charging and discharging power, “soc” is the state of charge and “cap” is the storage capacity. Conversion losses and losses over time proportional to the SoC may be considered. However, if conversion losses are integrated, then a binary variable is needed, which ensures that the storage may only charge or discharge at a certain time step in order to avoid the “annihilation” of energy by charging and discharging simultaneously. The variables may represent e.g. MW and MWh, kg/s and kg, or m³/s and m³, depending on the intended use.

#### PeriodStorage

This class can be used to model a long-term storage. To this end, the considered time horizon, e.g., a year, needs to be approximated by representative periods, which are directly assigned to the real periods. To this end, a period length of a day may be adequate. With the module “repr\_period” in the TES Gitlab group, such a direct assignment can be done. The model works just like the Storage class within the period, except that the “soc” variable is only relative. There is a second time scale with a time step of one period and a second state of charge “soc\_p”. Each representative period has always the same net charging or discharging. By direct assignment, this way the “soc\_p” over all periods can be define. The storage size is then defined by the maximum combination of “soc\_p” and the intraperiod charging/discharging.

#### StratifiedMultiLayer

This class represents a stratified water tank with multiple temperature layers. Each layer is modeled as ideal generic storage, similar to the “Storage” class. The indexed variables “m\_c” for charging, “m\_d” for discharging and “soc” for the state of charge have an additional dimension with the predefined temperature levels. The total mass in the stratified tank needs to be constant over time. I.e. if water is discharged from one temperature layer, the same amount of water has to be charged into the other layers. Due to the mass preservation constraint, the model cannot include conversion losses or losses over time proportional to the “Storage” class. However, it would be possible to integrate losses in such a form that over time water from hot layers transfers to colder layers. This is currently not integrated.

#### Demand

This class represents a general demand. The variable “d” represents the demand, which can be of any nature, e.g. MW, kg/s, m³/s, etc. Furthermore, the time series for the demand needs to be provided. The variable “d” is set to the given time series by a constraint.

**IMPORTANT:** If the DOOM is used in combination with HEN or scheduling models, then the demand variables “d” are the interfaces to these models and the constraint, which forces variable “d” to take values of the given time series needs to be deactivated.

#### HeatDemand

This class represents a heat demand, with a heat demand “q” and the corresponding mass flows “m\_in” and “m\_out”. There may be multiple inlet and outlet mass flows, in this case the indexed variables have an additional dimension accordingly. However, similar to the class “SolarThermal” the outlet mass flow distribution needs to be predefined.

**IMPORTANT:** If the DOOM is used in combination with HEN or scheduling models, then the demand variables “q”, or if necessary “m\_in” and “m\_out”, are the interfaces to these models and the constraint, which forces variable “d” to take values of the given time series needs to be deactivated.

#### Supply

This class represents a Supply with the variable “s”, e.g., for electricity of fuel in MWh, depending on the intended use. The price for the supply needs to be provided as time series. The supply has an objective term “energy”, which is the sum product of the supply “s” and the given price. Additionally, the maximum value of the variable “s” can be assigned with costs (power based costs) by the parameter “cost\_max\_s”.

#### DistrictHeating

#### This class represents district heating as supply. It works similar to the “Supply” class. However, the variable “q” is used for the heat supply, the variable m is used for the mass flow. Additionally, the maximum value of the variable “s” can be assigned with costs (power based costs) by the parameter “cost\_max\_q”.

#### Coupler

This unit represents a mass flow coupler. This unit was developed, but then discarded again, since its functionality can be achieved via a node. Maybe it can be extended and used in the future for some applications.

#### Node

This class represents a node for e.g. energy or mass balance equations. Though, the “Unit” class is the parent of this class, the node is not actually a unit. Nodes are only added to the “System” class, after all units are added, since the node accesses objects, which are generated by the attachment of units to the system. A node simply creates a constraint (the parameter “type” defines, if it is “==” , “>”, “<”, “>=”, or <=”). The dictionary “var\_in” defines the positive additions to the balance or the lhs, “var\_out” defines the negative additions or the rhs. “var\_in” and “var\_out” are lists of lists, which define the unit name, the unit port, and optionally a scalar multiplicator (which is by default 1). See “example\_case”. Furthermore, slack variables are added to the lhs and rhs, which are highly priced in the total objective. This way it is ensured that a system does not become infeasible by the interconnections of the units (and each unit should not cause infeasibilities, if the input parameters are set correctly).

### Auxiliary

In this module, multiple recurring functions are included. In the following, the important ones are described.

#### init\_uvwi\_param(unit)

This function sets the general unit parameters for “u\_active”, “v\_w\_active” and “i\_active” in dependence of the input parameters.

#### set\_general\_param(system, unit)

This function sets the parameters “depreciation\_period” and “interest\_rate” to the default system values, if not specified otherwise in the unit parametrization.

#### add\_var\_to\_model(system, unit)

This function adds the pyomo objects “Var” of a unit to the pyomo object “ConcreteModel” (which is named “model”).

#### add\_var\_uvwi(system, unit)

This function adds the commitment variable “u” to a unit, if the parameter “u\_active” is “True”, the startup/shutdown variables “v” and “w”, if the parameter “v\_w\_active” is “True”, and the investment binary variable “i”, if the parameter “i\_active” is “True”. Furthermore, if the unit parameter “exists” is present, then the variable “i” is set to 0 or 1 by constraint depending on whether the parameter “exists” is “False” or “True”, respectively.

#### add\_logic\_uvw(system, unit)

This function adds the logic constraint to a system, if both “u\_active” and “v\_w\_active” are “True”. Furthermore, the the minimum uptime and downtime constraints are added, see add\_min\_utdt(system, unit).

#### add\_simple\_m\_q(system, unit)

This function adds a constraint to the unit that defines the relation between a heat flow “q” and a mass flow “m” depending on the enthalpy of the inlet and outlet “h\_in” and “h\_out”, respectively.

#### add\_cap\_lim(unit, varname)

This function adds constraints for the capacity limit of a unit. The variables for the unit size is always named “cap”. However, in order to make model definition more clear, the parameter name is “cap\_” + some string, e.g. “cap\_q” for the capacity limits of the heat generation. Thus, this information needs to be added as “varname”. If the admissible size of a unit is semi-continuous, the variable “i” is needed.

#### add\_op\_lim(system, unit, varname)

This function adds constraints, which limit the minimum and maximum value of an operational variable, e.g., the heat generation “q” dependent on the unit size “cap”. The function input “varname” is the name of the operational variable, in the above case “q”. Dependening on the activity of “v” and “w” there are different formulations, which are selected automatically.

#### add\_ramp\_con(system, unit, varname)

This function adds constraint for the ramp limits of an operational variable, defined by “varname”. Dependent on the activity of “u”, “v” and “w” there are different formulations, which are selected automatically. The parameter “max\_susd” defines the maximum value of the operational variable after startup of before shutdown relative to the unit size variable “cap”.

#### add\_min\_utdt(system, unit)

This function adds constraints for the minimum uptime and downtime of a unit. This function is integrated in the function “add\_logic\_uvw”.

#### add\_lin\_dep(system, unit, varname)

This function adds a constraint, which defines the linear dependency of one operational variable of another operational variable. Dependening on the activity of “u” there are different formulations, which are selected automatically. If “u” is active, then the ratio of the two variables is not fixed, which is needed for varying efficiency for instance. The dependency is defined by the parameters “lim\_” + “varname”.

#### add\_es\_balance(system, unit, varname)

This function add a constraint for a storage balance. “varname” is a list, the first entry is the charging variable name, the second entry the discharging variable name, the third is the name of the SoC variable.

#### add\_obj\_inv(system, unit)

This function adds the objective for the investment costs. Dependening on the activity of “i” there are different formulations, which are selected automatically. If “i” is active, then fixed investment costs and thus economy of scale effects can be considered.

#### add\_obj\_u\_v\_w(system, unit)

This function adds general costs, which are frequently used for all units, such as fixed OPEX and startup and shutdown costs. Depending on the input parameters, the objectives are automatically generated. These objectives (“opex\_fix”, “cost\_SU”, “cost\_SD”) can be then added in the class description to the total objective of the class.

#### solve\_model(doom\_object, name)

Solves the pyomo model of the system object. The system parameter “opt” (dictionary) should be used to implement solver options. Currently, only the “timelimit” can be set.

#### save\_object(doom\_object, name)

This function saves the system object as pickle file.

#### import\_object(name)

This function imports a system object from a pickle file.

#### plot\_unit\_port(system)

Plots a figure with an axis for each unit. Each axis shows all ports of the corresponding unit. Mainly used for debugging and getting an overview.

#### plot\_node\_slack(system)

If any node slack variables are non-zero, then a figure with an axis for each node is plotted. The axis shows the rhs and lhs slack variable for each node.

#### plot\_full\_load\_hours(system, unit\_port)

Plots the full load hours (FLH) for each unit as bar plot. “unit\_port” defines the units and ports to be plotted. The FLH for solar units is defined by the usage of available solar energy. In other words, if a photovoltaic system generates as much power as possible with the available solar irradiation, then the FLH for this unit is 8760 h, so this value means, that no curtailment occurred.

#### plot\_unit\_sizes(system, unit)

Plots the unit sizes as bar plots. There will be four bar plots for energy conversion units (unit size = MW), storage units (unit size = MWh), stratified storages (unit size = m³) and solar units (unit size = m²). A dictionary “unit” with the entries “MW”, “MWh”, “m³” and “solar” needs to be provided, which include lists of the unit names.

#### plot\_strat\_soc(system, unit)

This function plots the SoC of a stratified storage. “unit” is the name of the stratified storage.

#### plot\_time\_series(system)

Plots all time series provided as input for the system.

#### plot\_unit\_inv\_cost(system, unit)

Plots the investment costs as bar plot for all units specified in “unit”.

#### plot\_energy\_provided(system, unit)

Plots a figure with three axis. The axis show heat, power and fuel provided/consumed for the units specified in “unit”. “unit” is a dictionary with the keys “q”, “p” and “f”, which contain lists of the corresponding unit names.

#### plot\_es\_soc(system)

Plots the SoC in MWh for all units, this is a custom function from a specific use case, but can be used in order to create further similar plots.

#### plot\_node(system)

Plots a figure an axis for each node with the rhs and lhs variables as stackplot.

#### plot\_heat\_demand(system)

Plots a figure with an axis for each “HeatDemand” unit with the variables “m\_in” and “m\_out”.