Recherche & Innovation

Holistic optimization framework for the operation of district cooling systems

Code technical documentation

Author: Chiam Zhong Lin

VEOLIA Recherche & Innovation SNC
Chemin de la Digue
78600 MAISONS-LAFFITTE • France
Tél. + 33 (0)1 34 93 31 31 • Fax : + 33 (0)1 34 93 31 10
www.veolia.com

Overview

This documentation describes the primary elements of the code which was used in the developing the hierarchical optimization framework for district cooling systems. The code has be modularized in to the following:

1. Mono-objective genetic algorithm (***.../vecmc_codes_zhonglin/ga/***)
2. Multi-objective genetic algorithm (***.../vecmc_codes_zhonglin/nsga_ii/***)
3. Mixed-integer linear program (***.../vecmc_codes_zhonglin/milp/***)
4. Reinforcement learning algorithm (***.../vecmc_codes_zhonglin/rl/***)
5. Sliding-window algorithm (***.../vecmc_codes_zhonglin/sw/***)

Modularization was done to facilitate the easy customization of future applications. The remaining part of the documentation will proceed to describe each of the following modules in the following manner:

1. Module purpose and dependencies.
2. Interaction of objects and functions.
3. Inputs and output formats.

Setting up the working environment

All codes in this document have been programmed in anaconda's distribution of python. To begin utilizing the framework the following two-step setup has to be done:

1. Clone relevant files from the github repository.
2. Setting up of a virtual working environment.
3. Optional: Setting up .bat file to run spyder in the correct environment.

1. Cloning file from the github repository

The github link is: https://github.com/zchiam002/vecmc_codes_zhonglin.git. This repository contains all the uncompiled source codes required for running/assembling the hierarchical optimization framework. The files can either be downloaded in a .zip format or cloned using a .git client such as git bash. Gitbash can be downloaded from https://git-scm.com/downloads.

2. Setting up of a virtual working environment

A copy of the working environment has been cloned to facilitate the setting up process. The file can be found in ***…\vecmc_codes_zhonglin\setup\c_nwk_main.yml***. To setup the virtual working environment, the following steps need to be done:

a. Open anaconda prompt in administrator mode.
b. Input: `conda --env create --n <new environment name> python = 3.5 --f = ...\vecmc_codes_zhonglin\setup\c_nwk_main.yml`

Doing so will create a new environment and install the relevant packages with the correct python version. To view the packages which have been installed:

a. Open anaconda prompt in administrator mode.
b. Activate the new environment by using the command: `activate <new environment name>`
c. Followed by `conda list`

3. Optional: Setting .bat file to run spyder in the correct environment

Assuming that spyder is the integrated development environment of choice, a sample .bat file can be found in the directory: ***.../vecmc_codes_zhonglin/setup/c_nwk_main_spyder_3_5.bat***. Open the file with notepad and edit the commands on the 2nd and 3rd lines:

a. 2nd line: Change to match the installation directory of the anaconda installation.
b. 3rd line: Change to match the chosen name of the new environment.

After this has been completed, the user will have everything that is required to run the codes which exist in the cloned repository.

Mono-objective genetic algorithm

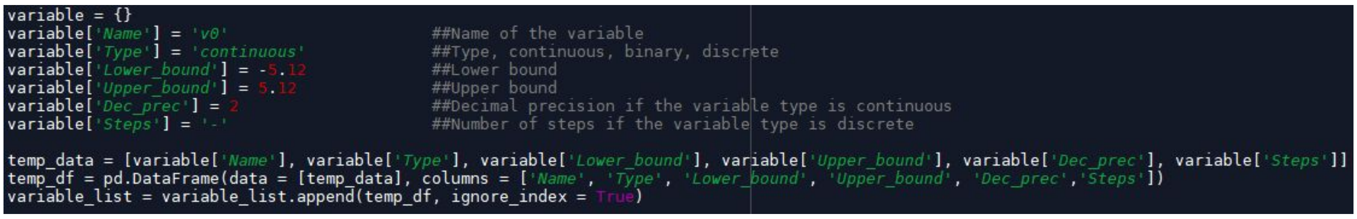
This is the vanilla implementation of the genetic algorithm which is designed for mono-objective optimization problems. The codes needed to execute this algorithm can be found in the directory: ***.../vecmc_codes_zhonglin/ga/***. A sample problem is included in the folder, hence it is possible to test the algorithm. This can be done by simply executing the python file: ***.../vecmc_codes_zhonglin/ga/ga_mono_run_nb.py***. The results of the run can be extracted from ***.../vecmc_codes_zhonglin/ga/ga_mono_results/best_obj_per_gen.csv***. The final row of the .csv file represents the best result.

1. Setting up the algorithm

Setting up this algorithm requires 2 files to be configured, the 1st allows hyperparameters (population size, iterations, etc.) and variables for optimization to be defined and the 2nd functions as a connector to other applications.

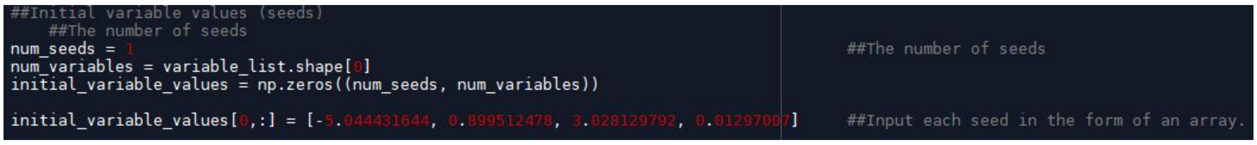
a. ***.../vecmc_codes_zhonglin/ga/ga_mono_simple_setup_nb.py***

Hyperparameters can be tuned under the function `ga_mono_simple_setup_nb`. Variables are defined in the function `variable_def`. In the latter, each of the defined variables are defined using dictionaries as shown in the following example:



```
variable = {}
variable['Name'] = 'v0'                          ##Name of the variable
variable['Type'] = 'continuous'                  ##Type, continuous, binary, discrete
variable['Lower_bound'] = -5.12                  ##Lower bound
variable['Upper_bound'] = 5.12                   ##Upper bound
variable['Dec_prec'] = 2                         ##Decimal precision if the variable type is continuous
variable['Steps'] = '-'                          ##Number of steps if the variable type is discrete

temp_data = [variable['Name'], variable['Type'], variable['Lower_bound'], variable['Upper_bound'], variable['Dec_prec'], variable['Steps']]
temp_df = pd.DataFrame(data = [temp_data], columns = ['Name', 'Type', 'Lower_bound', 'Upper_bound', 'Dec_prec','Steps'])
variable_list = variable_list.append(temp_df, ignore_index = True)
```

Additionally, to help the algorithm converge more quickly, initial seeds could also be added in the `variable_def` function as follows:



```
##Initial variable values (seeds)
    ##The number of seeds
num_seeds = 1                                                                   ##The number of seeds
num_variables = variable_list.shape[0]
initial_variable_values = np.zeros((num_seeds, num_variables))

initial_variable_values[0,:] = [-5.044431644, 0.899512478, 3.028129792, 0.01297007]    ##Input each seed in the form of an array.
```

b. ***.../vecmc_codes_zhonglin/ga/ga_mono_evaluate_objective_nb.py***

This python file contains the function `ga_mono_evaluate_objective_nb`. This function takes in two arguments `variable_values` and `iteration_number`. The `objective_value` is the output of this function.

i. `variable_values`: a list of length equivalent to the number of defined variables. These variables are meant to be fed into the user-defined function/application so that an objective function can be obtained.

ii. `iteration_number`: a unique integer which can be used to tag files if parallel processing is activated in the `ga_mono_simple_setup_nb` function. A simple use-case for this could be the writing of .csv files or reading of .json files specific for evaluating the `objective_value`.

iii. `objective_value`: a float variable which contains the value of the objective function evaluated, given the `variable_values`.

2. Running the algorithm

Once the hyperparameters, variables and the function/application to evaluate the objective function has been defined, the only thing left to do is to run the algorithm. This is done by executing the file: ***.../vecmc_codes_zhonglin/ga/run_mono_ga_nb.py*** in the spyder integrated development environment.

3. Extracting results

The results folder is located in: ***.../vecmc_codes_zhonglin/ga/ga_mono_results/***. This folder contains the following 3 files:

a. ***all_eval_pop.csv***
   Contains a record of all the variables which have been evaluated. The first and the last column represents the generation and the objective function respectively. Everything in the middle pertains to the variable values which have been evaluated.
b. ***best_agent_movement.csv***
   Contains a record of the movement of the best agent at the end of each generation - this means that if the best objective function value in the current generation is inferior to the last, the last will be recorded again. The last column records the objective function value. The rest of the columns record the value of the corresponding variable values.
c. ***best_obj_per_gen.csv***
   Contains a record of the movement of the best agent per generation. The first column records the generation number, last column the objective function value, and the rest of the columns, the corresponding value of the variables used for evaluation.

Multi-objective genetic algorithm

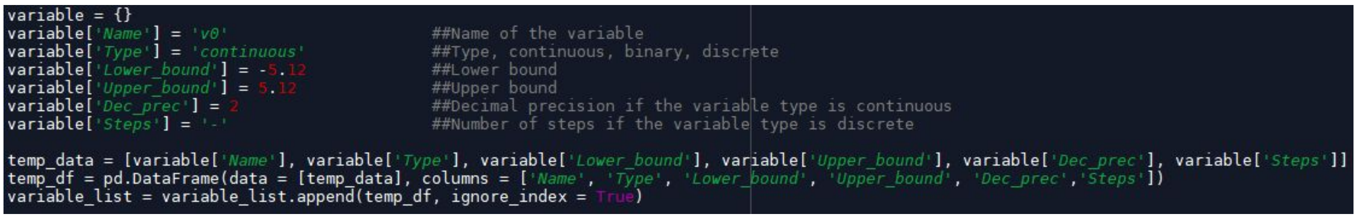
This algorithm is the python implementation of the second version of the non-dominated sort genetic algorithm (NSGA-II), which is designed for multi-objective optimization problems. The codes needed to execute this algorithm can be found in the directory: ***.../vecmc_codes_zhonglin/nsga_ii/***. A sample problem is included in the folder, hence it is possible to test the algorithm. This can be done by simply executing the python file: ***.../vecmc_codes_zhonglin/nsga_ii/nsga_ii_para_imple_simple_setup.py***. The results of the run can be extracted from the folder: ***.../vecmc_codes_zhonglin/nsga_ii/solution/***. There are 3 .csv files in this folder - representing the Pareto frontier objective functions (***solution.csv***), all evaluated variables (***solutionX.csv***) and the corresponding objective functions (***solutionObj.csv***) respectively.

1. Setting up the algorithm

Setting up this algorithm requires 2 files to be configured, the 1st allows hyperparameters (population size, iterations, etc.) and variables for optimization to be defined and the 2nd functions as a connector to other applications.

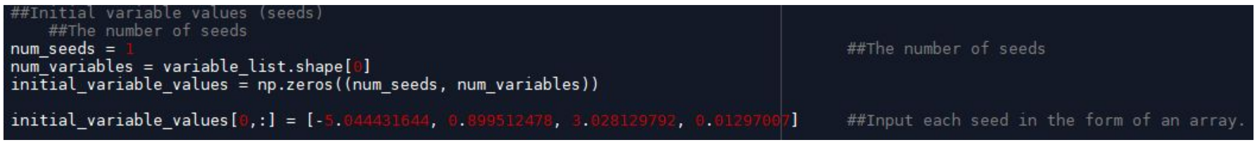
a. ***.../vecmc_codes_zhonglin/nsga_ii/nsga_ii_para_imple_simple_setup.py***

Hyperparameters can be tuned under the function `nsga_ii_para_imple_simple_setup`. Variables are defined in the function `variable_def`. In the latter, each of the defined variables are defined using dictionaries as shown in the following example:



```
variable = {}
variable['Name'] = 'v0'                         ##Name of the variable
variable['Type'] = 'continuous'                 ##Type, continuous, binary, discrete
variable['Lower_bound'] = -5.12                 ##Lower bound
variable['Upper_bound'] = 5.12                  ##Upper bound
variable['Dec_prec'] = 2                        ##Decimal precision if the variable type is continuous
variable['Steps'] = '-'                         ##Number of steps if the variable type is discrete

temp_data = [variable['Name'], variable['Type'], variable['Lower_bound'], variable['Upper_bound'], variable['Dec_prec'], variable['Steps']]
temp_df = pd.DataFrame(data = [temp_data], columns = ['Name', 'Type', 'Lower_bound', 'Upper_bound', 'Dec_prec','Steps'])
variable_list = variable_list.append(temp_df, ignore_index = True)
```

Additionally, to help the algorithm converge more quickly, initial seeds could also be added in the `variable_def` function as follows:



```
##Initial variable values (seeds)
    ##The number of seeds
num_seeds = 1                                                          ##The number of seeds
num_variables = variable_list.shape[0]
initial_variable_values = np.zeros((num_seeds, num_variables))

initial_variable_values[0,:] = [-5.044431644, 0.899512478, 3.028129792, 0.01297007]     ##Input each seed in the form of an array.
```

b. ***.../vecmc_codes_zhonglin/nsga_ii/nsga_ii_para_imple_evaluate_objective.py***

This python file contains the function `nsga_ii_para_imple_evaluate_objective`. This function takes in three arguments `variable_values`, `num_obj_func` and `iteration_number`. The `objective_value` is the output of this function.

i. `variable_values`: a list of length equivalent to the number of defined variables. These variables are meant to be fed into the user-defined function/application so that an objective function can be obtained.

ii. `num_obj_func`: an integer detailing the number of objective functions for which the function is to evaluated for.

iii. `iteration_number`: a unique integer which can be used to tag files if parallel processing is activated in the `nsga_ii_para_imple_simple_setup` function. A simple use-case for this could be the writing of .csv files or reading of .json files specific for evaluating the `objective_value`.

iv. `objective_value`: a list of objective function values evaluated for the given `variable_values`. The length of this list is equivalent to the value of `num_obj_func`.

2. Running the algorithm

Once the hyperparameters, variables and the function/application to evaluate the objective function has been defined, the only thing left to do is to run the algorithm. This is done by executing the file: ***.../vecmc_codes_zhonglin/nsga_ii/run_nsga_ii_para_imple.py*** in the spyder integrated development environment.

3. Extracting results

The results folder is located in: ***.../vecmc_codes_zhonglin/nsga_ii/solution/***. This folder contains the following 3 files:

a. ***solution.csv***

Contains a record of all the 'optimal' variable values and their corresponding objective function value which form the Pareto frontier. At the Pareto frontier, all solutions are considered to be 'equally good'. The user may then decide which solution is best suited for the use-case.

b. ***solutionObj.csv***

Contains a record of all evaluated objective functions which were evaluated during the process of deriving the Pareto frontier. Analysis of these values will give some insight into the parts of the solution space which was explored.

c. ***solutionX.csv***

Contains a record of the all variable values used to generate the objective functions found in ***solutionObj.csv***.

Mixed-integer linear program

The scripts needed to generate a mixed-integer linear program (MILP) which is compatible for the optimization of urban energy systems are described in this section. The python scripts will intake component-level models (chiller, network, etc.) and convert them into the .lp format. This file will subsequently be solved by invoking an external solver (GLPK).

The codes needed to execute this algorithm can be found in the directory: ***...vecmc_codes_zhonglin/milp/***. A sample problem is included in the folder, hence it is possible to test the algorithm. This can be done by simply executing the python file: ***.../vecmc_codes_zhonglin/milp/milp_run_script.py***. The results of the run can be extracted from the folder: ***.../vecmc_codes_zhonglin/milp/milp_results/***. There are 3 .csv files in this folder - representing the objective function (***results_obj_valueXX.csv***), continuous (***results_continuousXX.csv***) and binary variables (***results_binaryXX.csv***) respectively. The accompanying numbers (XX) are tagged to the end of each file as an identifier during parallel processing of information.

1. Setting up the algorithm

Setting up this algorithm requires 7 files to be configured. The process is rather complicated, hence it is explained using an example problem - a small district cooling system (DCS), with 2 chillers serving 2 customers. The following 2 figures describe the layout and corresponding models of the small DCS. The chillers, pumps, heat exchangers, evaporator and distribution networks are included in the model. For the chillers, only the evaporator side is considered. Constant flowrate and temperature values are used to represent the condenser side of the chiller. More detailed explanation of the chosen models and the corresponding derivation process can be found in the journal publication found in the following directory : ***...vecmc_codes_zhonglin\technical_documents\A hierarchical framework for holistic optimization of the operations of district cooling systems.pdf***.

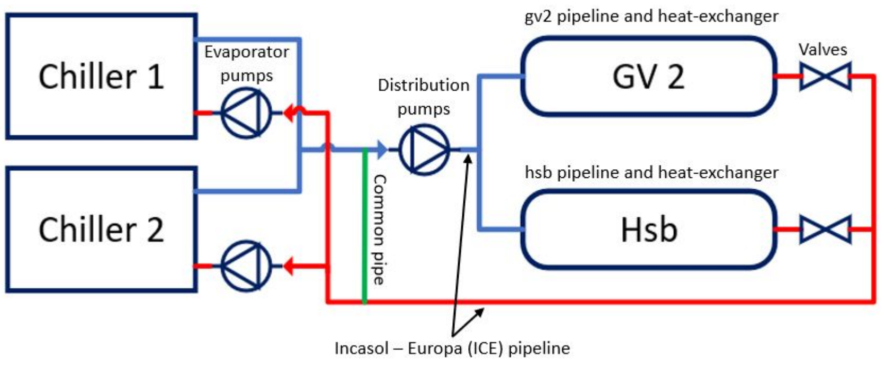

Chiller 1
Chiller 2
Evaporator pumps
Distribution pumps
Common pipe
gv2 pipeline and heat-exchanger
GV 2
Valves
hsb pipeline and heat-exchanger
Hsb
Incasol – Europa (ICE) pipeline

Figure 1: Sample MILP problem

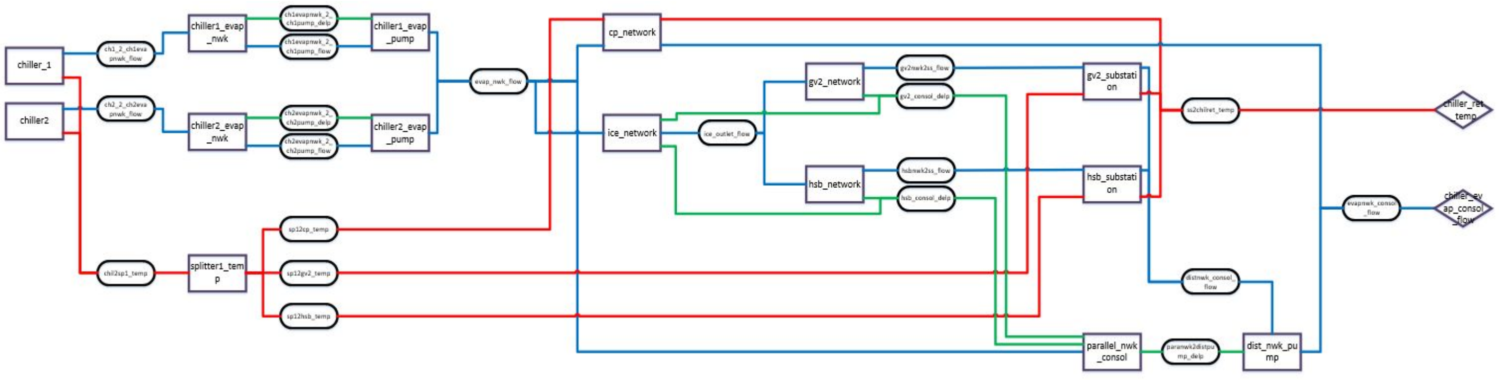


Figure 2: Corresponding MILP models of the sample problem

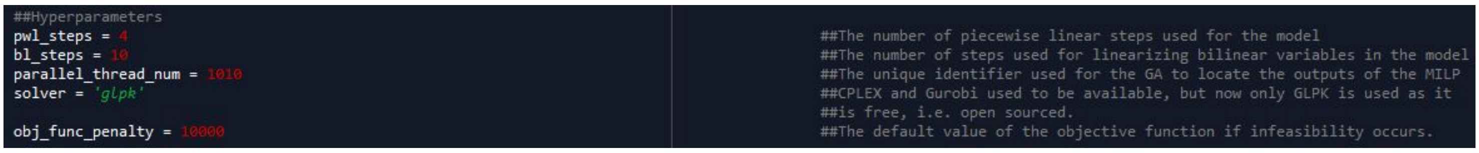
Details of the diagrams are found in the folder ***...vecmc_codes_zhonglin\milp\milp_models\***. The 3 relevant files are ***sample_model.vsdx***, ***sample_problem.vsdx*** and ***sample_model_labelled.pptx***.

a. ***...vecmc_codes_zhonglin\milp\ga_inputs\ga_inputs.csv***

This is a .csv file which simulates the output of the genetic algorithm (GA), which is to be treated as parameters in the MILP formulation, as described in the hierarchical framework described in ***...vecmc_codes_zhonglin\technical_documents\A hierarchical framework for holistic optimization of the operations of district cooling systems.pdf***.

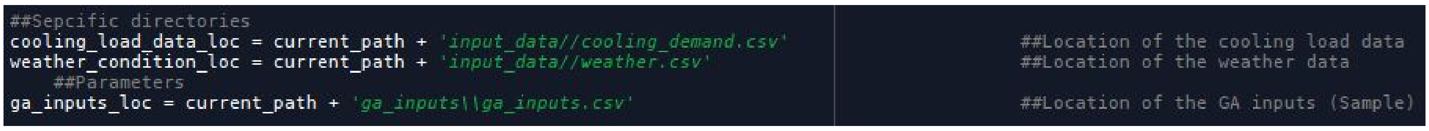
b. ***...vecmc_codes_zhonglin\milp\milp_run_script.py***

This file allows the hyperparameters, input file location (weather conditions, cooling demand, etc.) to be specified. The configurable hyperparameters are:



```
##Hyperparameters
pwl_steps = 4                                       ##The number of piecewise linear steps used for the model
bl_steps = 10                                       ##The number of steps used for linearizing bilinear variables in the model
parallel_thread_num = 1010                          ##The unique identifier used for the GA to locate the outputs of the MILP
solver = 'glpk'                                     ##CPLEX and Gurobi used to be available, but now only GLPK is used as it
                                                    ##is free, i.e. open sourced.
obj_func_penalty = 10000                            ##The default value of the objective function if infeasibility occurs.
```

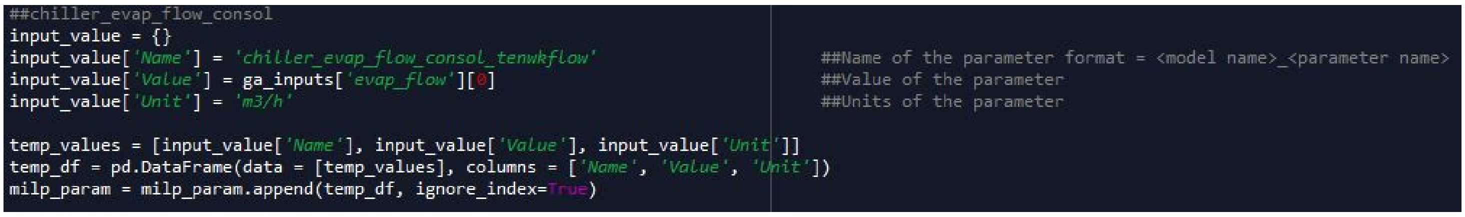
In the case of the sample problem, the input files required to run the MILP optimization model is:



```
##Sepcific directories
cooling_load_data_loc = current_path + 'input_data//cooling_demand.csv'        ##Location of the cooling load data
weather_condition_loc = current_path + 'input_data//weather.csv'               ##Location of the weather data
    ##Parameters
ga_inputs_loc = current_path + 'ga_inputs\\ga_inputs.csv'                      ##Location of the GA inputs (Sample)
```

c. ***...vecmc_codes_zhonglin\milp\mrs_manual_edit_milp_param.py***

This script contains the function `mrs_manual_edit_milp_param` which intakes hyperparameters and parameters and arranges them in the format which is readable by the component-level models (chillers, pumps, etc.). The following shows a sample entry of linking a specific parameter value and linking it to the corresponding model.



```
##chiller_evap_flow_consol
input_value = {}
input_value['Name'] = 'chiller_evap_flow_consol_tenwkflow'      ##Name of the parameter format = <model name>_<parameter name>
input_value['Value'] = ga_inputs['evap_flow'][0]                ##Value of the parameter
input_value['Unit'] = 'm3/h'                                    ##Units of the parameter

temp_values = [input_value['Name'], input_value['Value'], input_value['Unit']]
temp_df = pd.DataFrame(data = [temp_values], columns = ['Name', 'Value', 'Unit'])
milp_param = milp_param.append(temp_df, ignore_index=True)
```

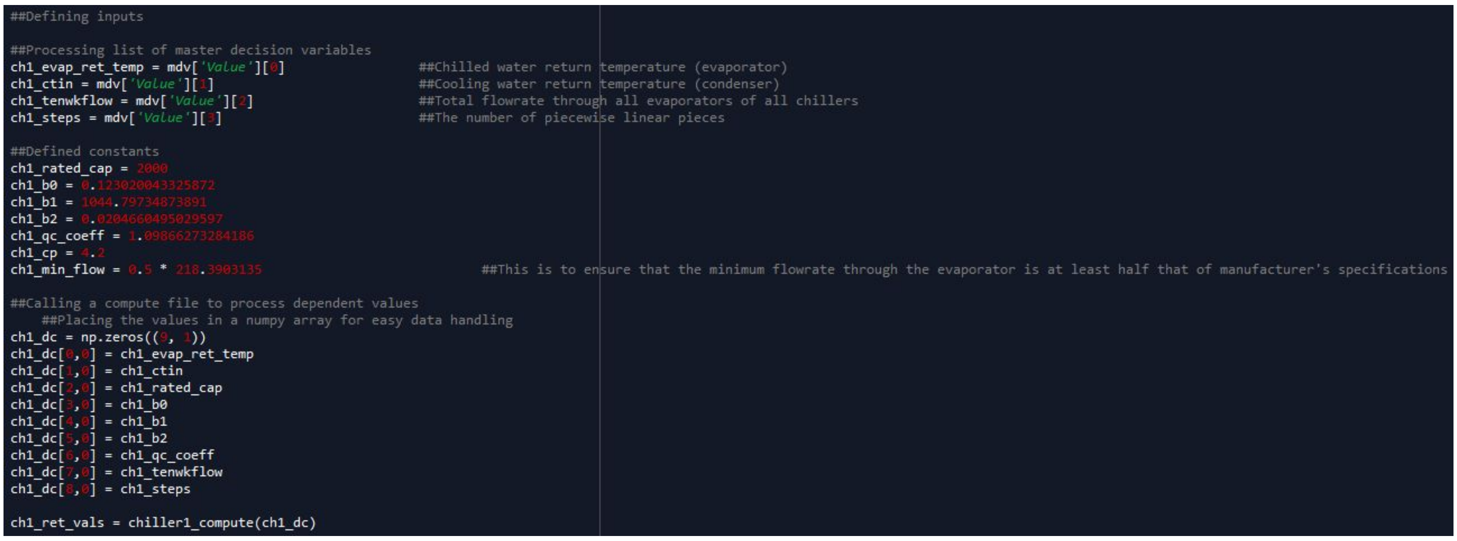
Special note has to be taken for the naming of `input_value['Name']`. The format to be respected is `<model name>_<parameter name>`. While the `<parameter name>` can be arbitrary,`<model name>` must correspond to the file name (without extension) of the corresponding model found in the folder ***...vecmc_codes_zhonglin\milp\milp_models\***.

d. ***...vecmc_codes_zhonglin\milp\milp_models\< input model name>.py***

This is an example python file of a model. The file ***chiller1.py*** shall be used for illustration purposes. The following describe the main components of the model file.

i. The function `checktype_<model name>`, return only 1 of the 3 strings - 'utility', 'process', or 'layer'. The 'process' type requires that the model cannot be 'deactivated' by the optimizer. The 'layer' type is only used by 1 file, which goes by the same name (see point (f)). Finally the 'utility' type is the most generic, which accepts all types of constraints. An example of the 'process' type can be seen in the file ***chiller_evap_flow_consol.py***. This model is introduced to ensure that the flowrate through the evaporator network must be equal to the value determined by the ***ga_input.csv*** file, which is a parameter to the sample problem.

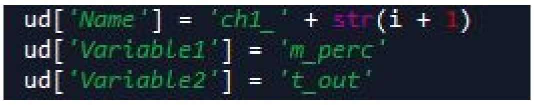
ii. Parameter values local to the model are defined here The first part involves the unpacking of the parameters, specific to this model defined in the function `mrs_manual_edit_milp_param`, followed by user-defined parameters. In the case of ***chiller1.py*** model, parameters such as the rated capacity and so on are defined. They are placed in a numpy array and sent into a `<model name>_compute` function for further processing. The `<model name>_compute` function is defined for the code readability - it is just a space of arbitrary computation of composite parameters. See ***chiller1_compute.py*** for an example. The `chiller1_compute` function was called to determine the linearized values of the chiller' Coefficient of Performance (COP). This is necessary as COP is a non-linear function and needs to be treated.



```
##Defining inputs

##Processing list of master decision variables
ch1_evap_ret_temp = mdv['Value'][0]                 ##Chilled water return temperature (evaporator)
ch1_ctin = mdv['Value'][1]                          ##Cooling water return temperature (condenser)
ch1_tenwkflow = mdv['Value'][2]                     ##Total flowrate through all evaporators of all chillers
ch1_steps = mdv['Value'][3]                         ##The number of piecewise linear pieces

##Defined constants
ch1_rated_cap = 2000
ch1_b0 = 0.123020043325872
ch1_b1 = 1044.79734873891
ch1_b2 = 0.0204660495029597
ch1_qc_coeff = 1.09866273284186
ch1_cp = 4.2
ch1_min_flow = 0.5 * 218.3903135                            ##This is to ensure that the minimum flowrate through the evaporator is at least half that of manufacturer's specifications

##Calling a compute file to process dependent values
    ##Placing the values in a numpy array for easy data handling
ch1_dc = np.zeros((9, 1))
ch1_dc[0,0] = ch1_evap_ret_temp
ch1_dc[1,0] = ch1_ctin
ch1_dc[2,0] = ch1_rated_cap
ch1_dc[3,0] = ch1_b0
ch1_dc[4,0] = ch1_b1
ch1_dc[5,0] = ch1_b2
ch1_dc[6,0] = ch1_qc_coeff
ch1_dc[7,0] = ch1_tenwkflow
ch1_dc[8,0] = ch1_steps

ch1_ret_vals = chiller1_compute(ch1_dc)
```

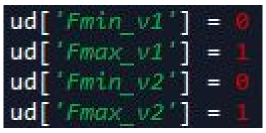
iii. In this subsection, the unit (chiller) model is defined. The unit definition is broken down to the following components:

- Name definition, where the model and variable names are given. This model format only supports up to 2 variables per model. If a variable is not used, it should be named '-'.



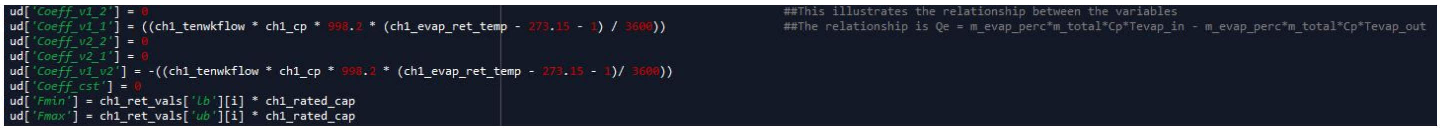
```
ud['Name'] = 'ch1_' + str(i + 1)
ud['Variable1'] = 'm_perc'
ud['Variable2'] = 't_out'
```

- Next the maximum and minimum values that the defined variables can undertake is determined. It is important to always keep in mind that the order of magnitudes of these values should be consistent across all models, or else the convergence issues may arise.



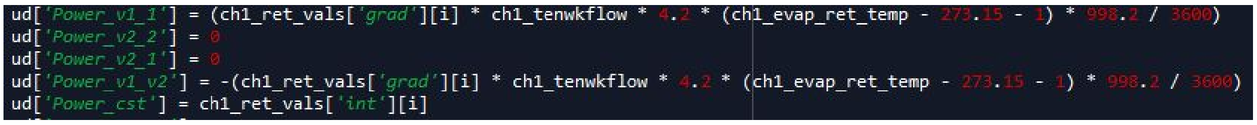
```
ud['Fmin_v1'] = 0
ud['Fmax_v1'] = 1
ud['Fmin_v2'] = 0
ud['Fmax_v2'] = 1
```

- This part defines an equation for which constraints have to be simultaneously placed on both defined variables. An example of this use-case is in the ***chiller1.py*** model, where the function $Q_e = f(variable1,\ variable2)$ used to describe the cooling output of the chiller, has to be bounded due to the rated capacity. `ud['Fmax']` and `ud['Fmin']` are used to constrain the values Q_e can take. `ud['Coeff_XXXX']` are used to define the coefficients of the equation. The general form is $y = A_0 variable_1^2 + A_1 variable_1 + A_2 variable_2^2 + A_3 variable_2 + A_4 variable_1 variable_2 + A_5$. The coefficients A_0 to A_5 correspond to `ud['Coeff_v1_2']` to `ud['Coeff_cst']` respectively.



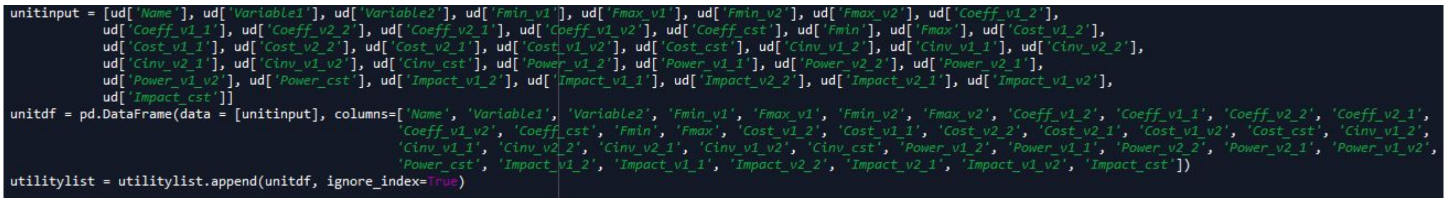
```
ud['Coeff_v1_2'] = 0
ud['Coeff_v1_1'] = ((ch1_tenwkflow * ch1_cp * 998.2 * (ch1_evap_ret_temp - 273.15 - 1) / 3600))
ud['Coeff_v2_2'] = 0
ud['Coeff_v2_1'] = 0
ud['Coeff_v1_v2'] = -((ch1_tenwkflow * ch1_cp * 998.2 * (ch1_evap_ret_temp - 273.15 - 1)/ 3600))
ud['Coeff_cst'] = 0
ud['Fmin'] = ch1_ret_vals['lb'][i] * ch1_rated_cap
ud['Fmax'] = ch1_ret_vals['ub'][i] * ch1_rated_cap

##This illustrates the relationship between the variables
##The relationship is Qe = m_evap_perc*m_total*Cp*Tevap_in - m_evap_perc*m_total*Cp*Tevap_out
```

- Next the contribution to the objective function is defined. There are 4 options - Cost, Cinv, Power and Impact which corresponds to operating cost, investment cost, power consumption and environmental impact respectively. Although more than 1 of these objective function equations can be filled up, only the 1 selected will be evaluated.



```
ud['Power_v1_1'] = (ch1_ret_vals['grad'][i] * ch1_tenwkflow * 4.2 * (ch1_evap_ret_temp - 273.15 - 1) * 998.2 / 3600)
ud['Power_v2_2'] = 0
ud['Power_v2_1'] = 0
ud['Power_v1_v2'] = -(ch1_ret_vals['grad'][i] * ch1_tenwkflow * 4.2 * (ch1_evap_ret_temp - 273.15 - 1) * 998.2 / 3600)
ud['Power_cst'] = ch1_ret_vals['int'][i]
```

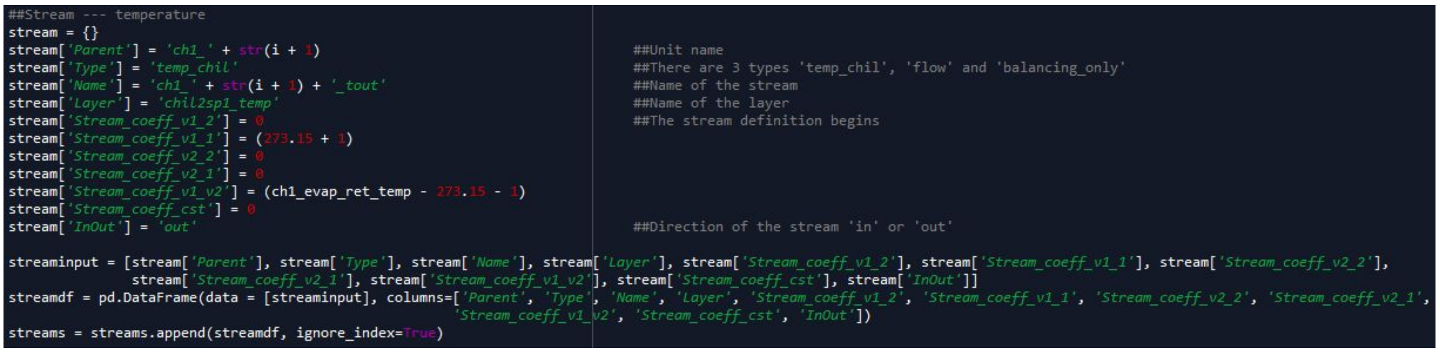
- Since the ***chiller1.py*** model is a 'utility' type the unit definition is appended to the `utilitylist` dataframe. Should it be of the 'process' type, it will be appended to the `processlist` dataframe.



```
unitinput = [ud['Name'], ud['Variable1'], ud['Variable2'], ud['Fmin_v1'], ud['Fmax_v1'], ud['Fmin_v2'], ud['Fmax_v2'], ud['Coeff_v1_2'],
             ud['Coeff_v1_1'], ud['Coeff_v2_2'], ud['Coeff_v2_1'], ud['Coeff_v1_v2'], ud['Coeff_cst'], ud['Fmin'], ud['Fmax'], ud['Cost_v1_2'],
             ud['Cost_v1_1'], ud['Cost_v2_2'], ud['Cost_v2_1'], ud['Cost_v1_v2'], ud['Cost_cst'], ud['Cinv_v1_2'], ud['Cinv_v1_1'], ud['Cinv_v2_2'],
             ud['Cinv_v2_1'], ud['Cinv_v1_v2'], ud['Cinv_cst'], ud['Power_v1_2'], ud['Power_v1_1'], ud['Power_v2_2'], ud['Power_v2_1'],
             ud['Power_v1_v2'], ud['Power_cst'], ud['Impact_v1_2'], ud['Impact_v1_1'], ud['Impact_v2_2'], ud['Impact_v2_1'], ud['Impact_v1_v2'],
             ud['Impact_cst']]
unitdf = pd.DataFrame(data = [unitinput], columns=['Name', 'Variable1', 'Variable2', 'Fmin_v1', 'Fmax_v1', 'Fmin_v2', 'Fmax_v2', 'Coeff_v1_2', 'Coeff_v1_1', 'Coeff_v2_2', 'Coeff_v2_1',
                                                   'Coeff_v1_v2', 'Coeff_cst', 'Fmin', 'Fmax', 'Cost_v1_2', 'Cost_v1_1', 'Cost_v2_2', 'Cost_v2_1', 'Cost_v1_v2', 'Cost_cst', 'Cinv_v1_2',
                                                   'Cinv_v1_1', 'Cinv_v2_2', 'Cinv_v2_1', 'Cinv_v1_v2', 'Cinv_cst', 'Power_v1_2', 'Power_v1_1', 'Power_v2_2', 'Power_v2_1', 'Power_v1_v2',
                                                   'Power_cst', 'Impact_v1_2', 'Impact_v1_1', 'Impact_v2_2', 'Impact_v2_1', 'Impact_v1_v2', 'Impact_cst'])
utilitylist = utilitylist.append(unitdf, ignore_index=True)
```

- The `for` loop is used to wrap the unit definition so that several sub-models are also defined. This is only done when necessary - for the case of ***chiller1.py***, the piecewise linearization of the COP necessitates that several models (as many as the number of linear pieces) be defined, each model using a fixed value of the COP.

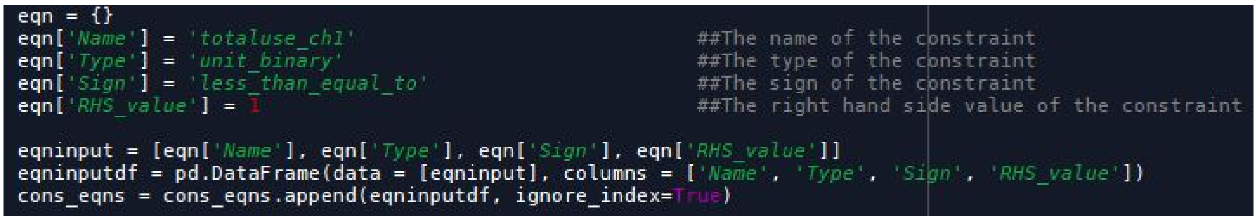
iv. After the unit definition is completed, the streams are defined. Streams denote the flows (fluid, temperature, pressure, etc.) in and out of the unit and are defined in the following format.



```
##Stream --- temperature
stream = {}
stream['Parent'] = 'ch1_' + str(i + 1)                              ##Unit name
stream['Type'] = 'temp_chil'                                        ##There are 3 types 'temp_chil', 'flow' and 'balancing_only'
stream['Name'] = 'ch1_' + str(i + 1) + '_tout'                      ##Name of the stream
stream['Layer'] = 'chil2sp1_temp'                                   ##Name of the layer
stream['Stream_coeff_v1_2'] = 0                                     ##The stream definition begins
stream['Stream_coeff_v1_1'] = (273.15 + 1)
stream['Stream_coeff_v2_2'] = 0
stream['Stream_coeff_v2_1'] = 0
stream['Stream_coeff_v1_v2'] = (ch1_evap_ret_temp - 273.15 - 1)
stream['Stream_coeff_cst'] = 0
stream['InOut'] = 'out'                                             ##Direction of the stream 'in' or 'out'

streaminput = [stream['Parent'], stream['Type'], stream['Name'], stream['Layer'], stream['Stream_coeff_v1_2'], stream['Stream_coeff_v1_1'], stream['Stream_coeff_v2_2'],
               stream['Stream_coeff_v2_1'], stream['Stream_coeff_v1_v2'], stream['Stream_coeff_cst'], stream['InOut']]
streamdf = pd.DataFrame(data = [streaminput], columns=['Parent', 'Type', 'Name', 'Layer', 'Stream_coeff_v1_2', 'Stream_coeff_v1_1', 'Stream_coeff_v2_2', 'Stream_coeff_v2_1',
                                                       'Stream_coeff_v1_v2', 'Stream_coeff_cst', 'InOut'])
streams = streams.append(streamdf, ignore_index=True)
```

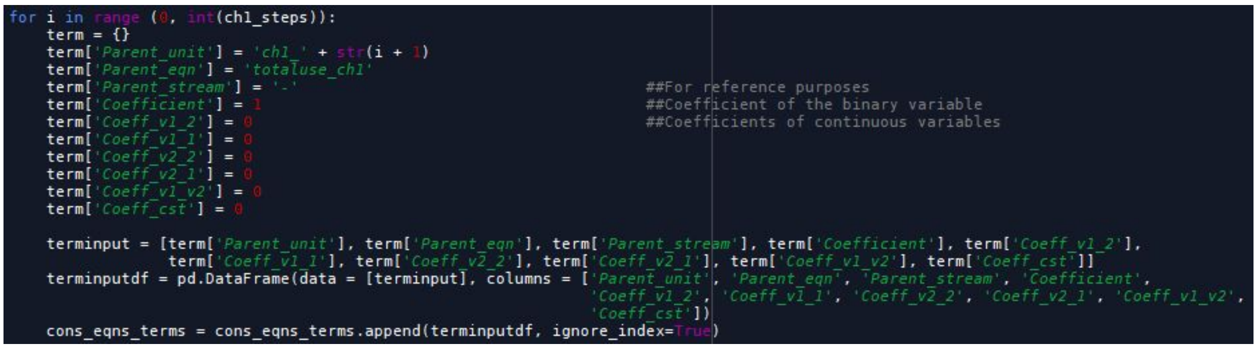
There are 4 types of values `stream['Type']` can take - 'temp_chil', 'pressure', 'flow' and 'balancing_only' which corresponds to '<=', '<=', '>=' and '=' respectively. The above example of the stream definition shows that outbound stream (`stream['Inout'] = 'out'`) have to be less than the inbound stream(s) belonging to the same `stream['Layer']`.

v. Finally additional constraints are defined. These constraints are designed to execute 2 features - to impose limits on the number of units activated, and upper and lower bounds of streams. There are 2 main aspects to defining the constraint - the main constraint setup and the components of the constraint. The following example demonstrates how to set up a constraint.



```
eqn = {}
eqn['Name'] = 'totaluse_ch1'                              ##The name of the constraint
eqn['Type'] = 'unit_binary'                               ##The type of the constraint
eqn['Sign'] = 'less_than_equal_to'                        ##The sign of the constraint
eqn['RHS_value'] = 1                                      ##The right hand side value of the constraint

eqninput = [eqn['Name'], eqn['Type'], eqn['Sign'], eqn['RHS_value']]
eqninputdf = pd.DataFrame(data = [eqninput], columns = ['Name', 'Type', 'Sign', 'RHS_value'])
cons_eqns = cons_eqns.append(eqninputdf, ignore_index=True)
```

There are 2 types of values that `eqn['Type']` can undertake - 'unit binary' and 'stream_mimit_modified'. The former is strictly used for binary values, while the latter for continuous values. There are 3 possible values that `eqn['Sign']` can assume - 'less_than_equal_to', 'equal_to' and 'greater_than_equal_to'. Finally the right hand side value of the constraint is defined in `eqn['RHS_value']`. The following figure illustrates how to populate the left-hand-side of the equation.



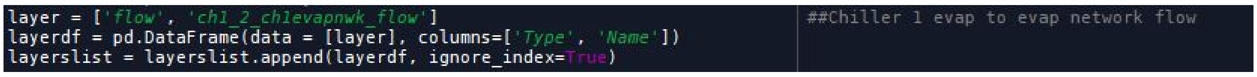
```
for i in range (0, int(ch1_steps)):
    term = {}
    term['Parent_unit'] = 'ch1_' + str(i + 1)
    term['Parent_eqn'] = 'totaluse_ch1'
    term['Parent_stream'] = '-'                                  ##For reference purposes
    term['Coefficient'] = 1                                      ##Coefficient of the binary variable
    term['Coeff_v1_2'] = 0                                       ##Coefficients of continuous variables
    term['Coeff_v1_1'] = 0
    term['Coeff_v2_2'] = 0
    term['Coeff_v2_1'] = 0
    term['Coeff_v1_v2'] = 0
    term['Coeff_cst'] = 0

    terminput = [term['Parent_unit'], term['Parent_eqn'], term['Parent_stream'], term['Coefficient'], term['Coeff_v1_2'],
                 term['Coeff_v1_1'], term['Coeff_v2_2'], term['Coeff_v2_1'], term['Coeff_v1_v2'], term['Coeff_cst']]
    terminputdf = pd.DataFrame(data = [terminput], columns = ['Parent_unit', 'Parent_eqn', 'Parent_stream', 'Coefficient',
                                                              'Coeff_v1_2', 'Coeff_v1_1', 'Coeff_v2_2', 'Coeff_v2_1', 'Coeff_v1_v2',
                                                              'Coeff_cst'])
    cons_eqns_terms = cons_eqns_terms.append(terminputdf, ignore_index=True)
```

Firstly, value of the `term['Parent_unit']` has to be identical to `ud['Name']` found in the unit definition (part d. ii., point 1). This is to relate the constraint term to the relevant unit. Secondly, `term['Parent_eqn']` has to be identical to the value defined in `eqn['Name']` , this is to relate the term defined to the right-hand-side of the equation. Next the coefficients of the right-hand-side terms are defined. `term['Coefficient']` defines the binary term and `term['Coeff_v1_2']` to `term['Coeff_cst']` defines the continuous terms.

Assuming that the value of ch1_steps = 4, in the following example, the resultant constraint defined will be $y_{ch1\_0} + y_{ch1\_1} + y_{ch1\_2} + y_{ch1\_3} \leq 1$, where y_i are binary variables. The reason that this constraint is defined is to ensure that only 1 part of the discretized chiller unit is activated. Continuous constraints are defined in the same manner.

e. ***...vecmc_codes_zhonglin\milp\milp_models\layers.py***

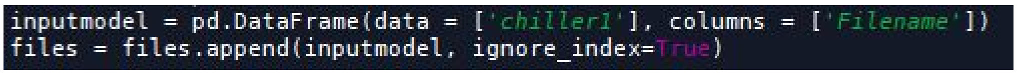
This file contains a record of all the ‘layers’ which connects all the ‘streams’ defined in all the sub-models. In this file, both the name and the type of each stream is defined.



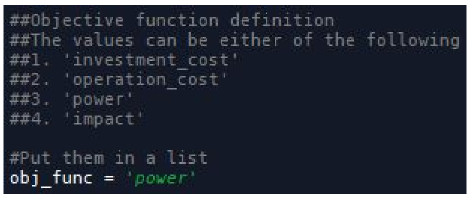
```
layer = ['flow', 'ch1_2_chlevapnwk_flow']                              ##Chiller 1 evap to evap network flow
layerdf = pd.DataFrame(data = [layer], columns=['Type', 'Name'])
layerslist = layerslist.append(layerdf, ignore_index=True)
```

f. ***...vecmc_codes_zhonglin\milp\milp_models\milp_prog_run.py***

This file contains the names of the models defined (excluding _compute.py files). It is also where the objective function (`‘obj_func’`) is defined. The objective function can undertake either of 4 values: cost, cinv, power and impact, which corresponds to operating cost, investment cost, power consumption and environmental impact respectively. The definition here will determine which one of the coefficients the solver will evaluate in the unit definition found in d. iii. point 3. For instance if `obj_func` = ‘power’, all `ud[‘Power_’]` terms will be evaluated.



```
inputmodel = pd.DataFrame(data = ['chiller1'], columns = ['Filename'])
files = files.append(inputmodel, ignore_index=True)
```



```
##Objective function definition
##The values can be either of the following
##1. 'investment_cost'
##2. 'operation_cost'
##3. 'power'
##4. 'impact'

#Put them in a list
obj_func = 'power'
```

2. Running the algorithm

Once the hyperparameters, variables and the function/application to evaluate the objective function has been defined, the only thing left to do is to run the algorithm. This is done by executing the file: ***.../vecmc_codes_zhonglin/milp/milp_run_script.py*** in the spyder integrated development environment.

3. Extracting results

The results folder is located in: ***.../vecmc_codes_zhonglin/milp/milp_results/***. This folder contains the following 3 files:

a. ***results_obj_value.csv***

Contains the value of the objective function after the model has been solved. if the value is ‘na’, this means that the model defined is invalid.

b. ***results_binary.csv***

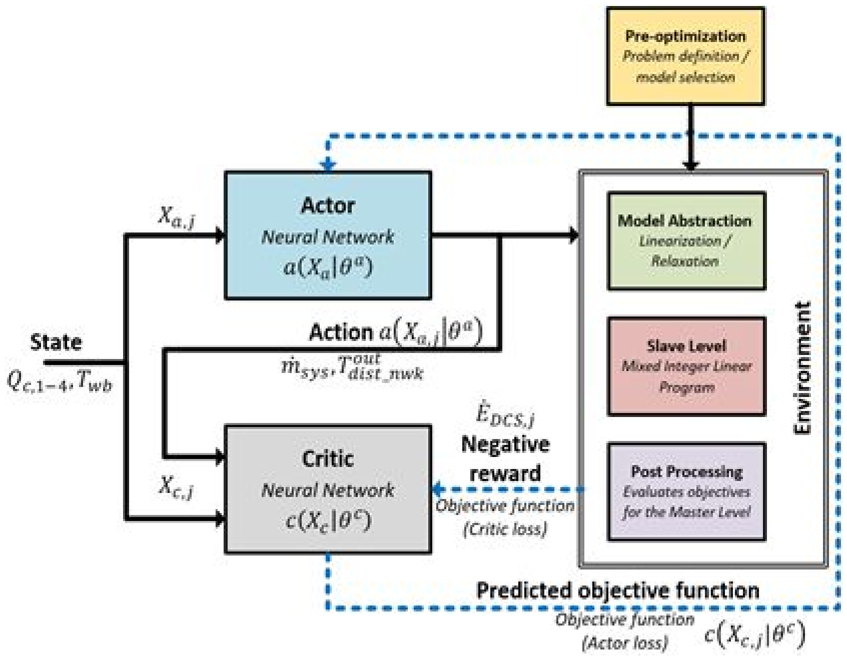
Contains the corresponding value of all the binary variables at the optimum. If the value is 1, it means that the unit has been activated. 0 means otherwise.

c. ***results_continuous.csv***

Contains a record of all the values of the continuous variables at the optimum. These values should be in the range of `ud[‘Fmin’]` and `ud[‘Fmax’]` for each model defined.

Reinforcement learning algorithm

The scripts needed to generate a reinforcement-learner coupled with a mixed-integer linear program (RL-MILP) for the purpose of optimizing the operations of urban energy systems are described in this section. The RL algorithm is a modified version of the actor-critic algorithm compatible for state-independent problems. The following figure summarizes the architecture of the state-independent RL-MILP algorithm, where the environment is the MILP problem.



Greater details of the implementation and the problem this algorithm tries to solve can be found in the conference paper published, which is located in the following directory: ***...vecmc_codes_zhonglin\technical_documents\ICAE_2019_RL_MILP_PAPER_FINAL.pdf***. Two sample codes with test problems are also included to facilitate the understanding of reinforcement-learning in general. The description of the sample cases are:

a. Contextual bandit problem, using policy-gradient algorithm: ***...vecmc_codes_zhonglin\rl\test_codes\contextual_bandit_tf_implementation.pdf***.
b. Continuous mountain car problem, using state-dependent actor-critic algorithm: ***...vecmc_codes_zhonglin\rl\test_codes\continuous_mountain_car_actor_critic.pdf***.

These two problems are included to aid the user's understanding of the difference in reinforcement-learning architectures as well as state-dependence/independence.

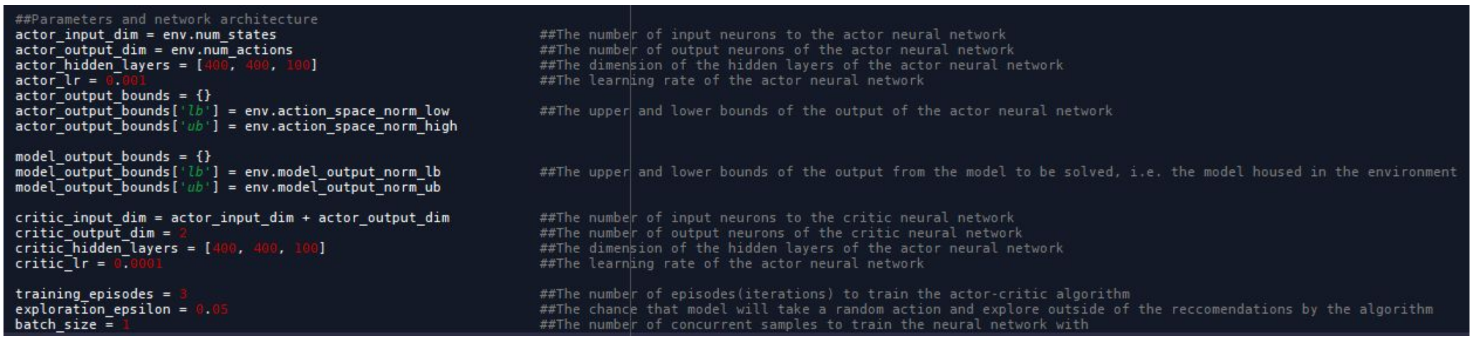
The codes needed to execute this algorithm can be found in the directory: ***...vecmc_codes_zhonglin/rl/***. A sample problem is included in the folder, hence it is possible to test the algorithm. This can be done by simply executing the python file: ***.../vecmc_codes_zhonglin/rl/main_function.py***. The resultant model generated from the run can be accessed from the folder: ***.../vecmc_codes_zhonglin/rl/saved_model/***.

1. Setting up the algorithm

Setting up this algorithm requires 2 files to be configured, the 1st allows the architecture of the neural-networks and the training parameters to be defined and the 2nd functions as a connector to the model to be solved/learned.

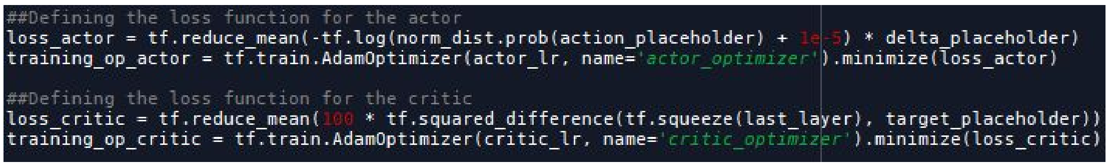
a. ***…\vecmc_codes_zhonglin\rl\main_function.py***

   This file houses the main function (`main_a2c_function`) which defines the parameters and the flow of the algorithm. The following diagram illustrates the setup of the training parameters and the architecture of the actor and critic neural-network.



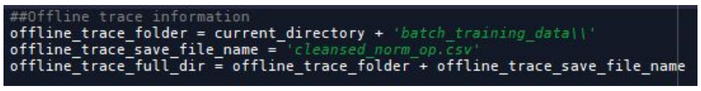
```
##Parameters and network architecture
actor_input_dim = env.num_states                                    ##The number of input neurons to the actor neural network
actor_output_dim = env.num_actions                                  ##The number of output neurons of the actor neural network
actor_hidden_layers = [400, 400, 100]                               ##The dimension of the hidden layers of the actor neural network
actor_lr = 0.001                                                    ##The learning rate of the actor neural network
actor_output_bounds = {}
actor_output_bounds['lb'] = env.action_space_norm_low               ##The upper and lower bounds of the output of the actor neural network
actor_output_bounds['ub'] = env.action_space_norm_high

model_output_bounds = {}
model_output_bounds['lb'] = env.model_output_norm_lb                ##The upper and lower bounds of the output from the model to be solved, i.e. the model housed in the environment
model_output_bounds['ub'] = env.model_output_norm_ub

critic_input_dim = actor_input_dim + actor_output_dim               ##The number of input neurons to the critic neural network
critic_output_dim = 2                                               ##The number of output neurons of the critic neural network
critic_hidden_layers = [400, 400, 100]                              ##The dimension of the hidden layers of the actor neural network
critic_lr = 0.0001                                                  ##The learning rate of the actor neural network

training_episodes = 3                                               ##The number of episodes(iterations) to train the actor-critic algorithm
exploration_epsilon = 0.05                                          ##The chance that model will take a random action and explore outside of the reccomendations by the algorithm
batch_size = 1                                                      ##The number of concurrent samples to train the neural network with
```

After the declaration of the parameters and the network architecture, the optimization algorithm for the weights of the neural-networks have to be defined. They can be done in the section of the code illustrated in the following figure.

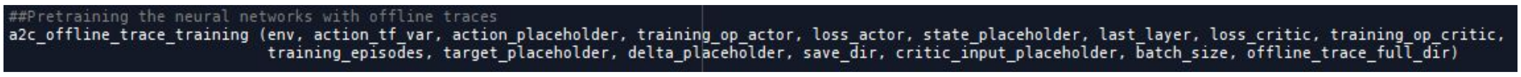


```
##Defining the loss function for the actor
loss_actor = tf.reduce_mean(-tf.log(norm_dist.prob(action_placeholder) + 1e-5) * delta_placeholder)
training_op_actor = tf.train.AdamOptimizer(actor_lr, name='actor_optimizer').minimize(loss_actor)

##Defining the loss function for the critic
loss_critic = tf.reduce_mean(100 * tf.squared_difference(tf.squeeze(last_layer), target_placeholder))
training_op_critic = tf.train.AdamOptimizer(critic_lr, name='critic_optimizer').minimize(loss_critic)
```

Next, the user has the option of pre-training the model with offline data traces. These data traces are often data that are high in quality, examples which serve to help the model converge more quickly. In this example, the RL model has the option of being pre-trained with optimal outputs from the genetic algorithm. The following figures illustrate the sections of the code which is associated with this feature. If this is not required, these sections should be commented.

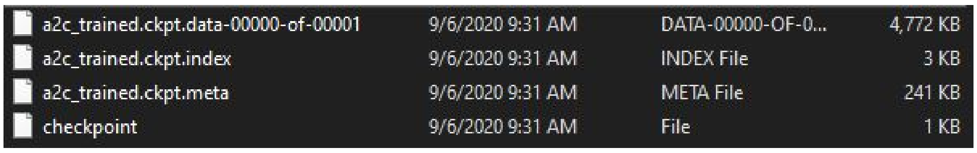


```
##Offline trace information
offline_trace_folder = current_directory + 'batch_training_data\\'
offline_trace_save_file_name = 'cleansed_norm_op.csv'
offline_trace_full_dir = offline_trace_folder + offline_trace_save_file_name
```



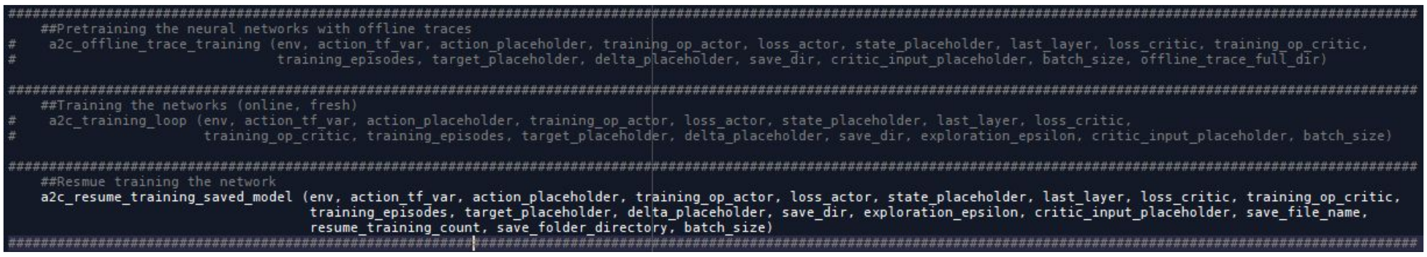
```
##Pretraining the neural networks with offline traces
a2c_offline_trace_training (env, action_tf_var, action_placeholder, training_op_actor, loss_actor, state_placeholder, last_layer, loss_critic, training_op_critic,
                                 training_episodes, target_placeholder, delta_placeholder, save_dir, critic_input_placeholder, batch_size, offline_trace_full_dir)
```

Finally, should the user feel the need to improve the performance of existing models, additional training iterations can be performed on saved models. To do this, the user has to first ensure that the 4 components of the saved model are placed in the location ***...vecmc_codes_zhonglin\rl\saved_model\*** with the default names as shown in the following figure.



```
a2c_trained.ckpt.data-00000-of-00001   9/6/2020 9:31 AM   DATA-00000-OF-0...   4,772 KB
a2c_trained.ckpt.index                 9/6/2020 9:31 AM   INDEX File               3 KB
a2c_trained.ckpt.meta                  9/6/2020 9:31 AM   META File              241 KB
checkpoint                             9/6/2020 9:31 AM   File                     1 KB
```

Subsequently, the appropriate lines in the function has to be commented/uncommented exactly as shown:

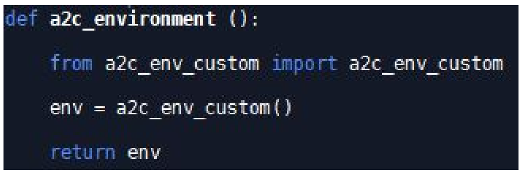


```
########################################################################################################################
    ##Pretraining the neural networks with offline traces
#    a2c_offline_trace_training (env, action_tf_var, action_placeholder, training_op_actor, loss_actor, state_placeholder, last_layer, loss_critic, training_op_critic,
#                                training_episodes, target_placeholder, delta_placeholder, save_dir, critic_input_placeholder, batch_size, offline_trace_full_dir)

########################################################################################################################
    ##Training the networks (online, fresh)
#    a2c_training_loop (env, action_tf_var, action_placeholder, training_op_actor, loss_actor, state_placeholder, last_layer, loss_critic,
#                       training_op_critic, training_episodes, target_placeholder, delta_placeholder, save_dir, exploration_epsilon, critic_input_placeholder, batch_size)

########################################################################################################################
    ##Resmue training the network
    a2c_resume_training_saved_model (env, action_tf_var, action_placeholder, training_op_actor, loss_actor, state_placeholder, last_layer, loss_critic, training_op_critic,
                                     training_episodes, target_placeholder, delta_placeholder, save_dir, exploration_epsilon, critic_input_placeholder, save_file_name,
                                     resume_training_count, save_folder_directory, batch_size)
########################################################################################################################
```

The improved model as a result of the additional training will be found in the directory ***...vecmc_codes_zhonglin\rl\saved_model\resume_XX_starting_point***\, where XX is the value given to the `resume_training_count` variable found in the `main_a2c_function` function.

b. ***…\vecmc_codes_zhonglin\rl\a2c_environment.py***

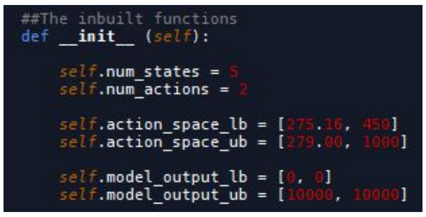
This is where the user has to specify the problem to be solved. In the following example, the `a2c_env_custom` class houses the environment of the problem to be solved.



```
def a2c_environment ():

    from a2c_env_custom import a2c_env_custom

    env = a2c_env_custom()

    return env
```

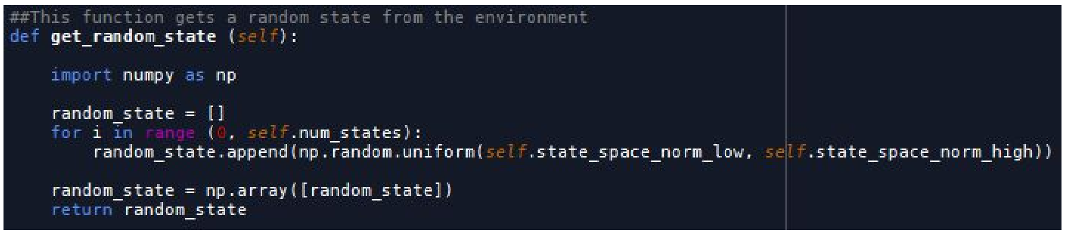
To define a model which is compatible with this implementation of the RL algorithm, the following features need to be present:

i. Number of actions, states and the bounds of the environment.



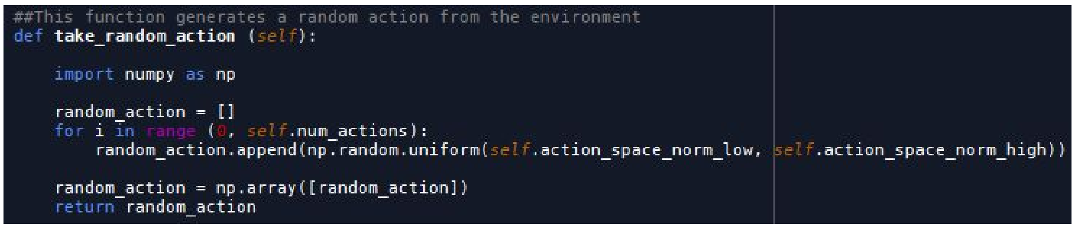
```
##The inbuilt functions
def __init__ (self):

    self.num_states = 5
    self.num_actions = 2

    self.action_space_lb = [275.16, 450]
    self.action_space_ub = [279.00, 1000]

    self.model_output_lb = [0, 0]
    self.model_output_ub = [10000, 10000]
```

ii. The `get_random_state` function for the model for the RL algorithm to randomly take states from the environment.



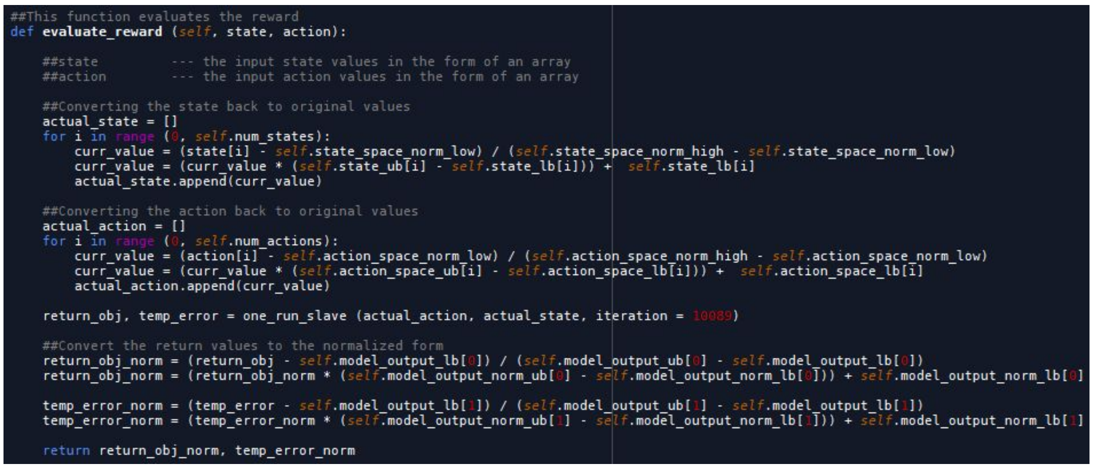
```
##This function gets a random state from the environment
def get_random_state (self):

    import numpy as np

    random_state = []
    for i in range (0, self.num_states):
        random_state.append(np.random.uniform(self.state_space_norm_low, self.state_space_norm_high))

    random_state = np.array([random_state])
    return random_state
```

iii. The `take_random_action` function for the model to randomly take an action in the environment.



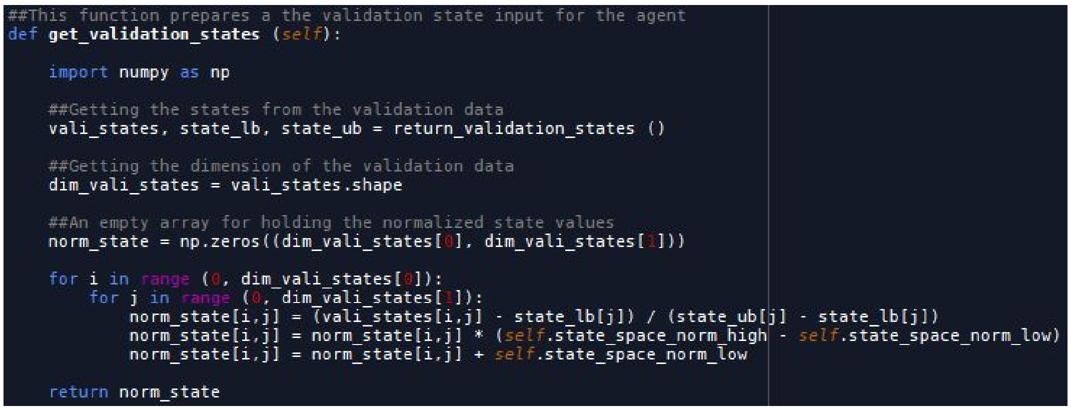
```
##This function generates a random action from the environment
def take_random_action (self):

    import numpy as np

    random_action = []
    for i in range (0, self.num_actions):
        random_action.append(np.random.uniform(self.action_space_norm_low, self.action_space_norm_high))

    random_action = np.array([random_action])
    return random_action
```

iv. And a means for the algorithm to get a reward from the environment for taking an action.



```
##This function evaluates the reward
def evaluate_reward (self, state, action):

    ##state          --- the input state values in the form of an array
    ##action         --- the input action values in the form of an array

    ##Converting the state back to original values
    actual_state = []
    for i in range (0, self.num_states):
        curr_value = (state[i] - self.state_space_norm_low) / (self.state_space_norm_high - self.state_space_norm_low)
        curr_value = (curr_value * (self.state_ub[i] - self.state_lb[i])) +  self.state_lb[i]
        actual_state.append(curr_value)

    ##Converting the action back to original values
    actual_action = []
    for i in range (0, self.num_actions):
        curr_value = (action[i] - self.action_space_norm_low) / (self.action_space_norm_high - self.action_space_norm_low)
        curr_value = (curr_value * (self.action_space_ub[i] - self.action_space_lb[i])) +  self.action_space_lb[i]
        actual_action.append(curr_value)

    return_obj, temp_error = one_run_slave (actual_action, actual_state, iteration = 10089)

    ##Convert the return values to the normalized form
    return_obj_norm = (return_obj - self.model_output_lb[0]) / (self.model_output_ub[0] - self.model_output_lb[0])
    return_obj_norm = (return_obj_norm * (self.model_output_norm_ub[0] - self.model_output_norm_lb[0])) + self.model_output_norm_lb[0]

    temp_error_norm = (temp_error - self.model_output_lb[1]) / (self.model_output_ub[1] - self.model_output_lb[1])
    temp_error_norm = (temp_error_norm * (self.model_output_norm_ub[1] - self.model_output_norm_lb[1])) + self.model_output_norm_lb[1]

    return return_obj_norm, temp_error_norm
```

v. Finally, the specification of the location of the validation dataset. This is used to evaluate the model once the training is deemed to be sufficient.



```
##This function prepares a the validation state input for the agent
def get_validation_states (self):

    import numpy as np

    ##Getting the states from the validation data
    vali_states, state_lb, state_ub = return_validation_states ()

    ##Getting the dimension of the validation data
    dim_vali_states = vali_states.shape

    ##An empty array for holding the normalized state values
    norm_state = np.zeros((dim_vali_states[0], dim_vali_states[1]))

    for i in range (0, dim_vali_states[0]):
        for j in range (0, dim_vali_states[1]):
            norm_state[i,j] = (vali_states[i,j] - state_lb[j]) / (state_ub[j] - state_lb[j])
            norm_state[i,j] = norm_state[i,j] * (self.state_space_norm_high - self.state_space_norm_low)
            norm_state[i,j] = norm_state[i,j] + self.state_space_norm_low

    return norm_state
```

It is to be emphasized that the 5 above-mentioned features have to be present for the proper definition of any custom class which houses the environment. That said, the user is free to modify the contents within the above declarations/functions in any way suitable. The above figures were extracted from the file ***...vecmc_codes_zhonglin\rl\a2c_env_custom.py***, which houses the MILP problem described in the published conference paper. This is just a test problem which should be modified to suit the needs of the user.

2. Running the algorithm

Once the parameters, neural-network architecture and the environment class has been defined, the only thing left to do is to run the algorithm. This is done by executing the file: ***.../vecmc_codes_zhonglin/rl/main_function.py*** in the spyder integrated development environment.

3. Extracting results

The trained model can be extracted from the location ***...vecmc_codes_zhonglin\rl\saved_model\***. Additionally, the algorithm is configured to also save the model at every 100 iteration interval. The saved model, with the corresponding loss statistics can be found in sub-folders in the directory ***...vecmc_codes_zhonglin\rl\saved_model\save_iteration_XX***, where XX - 1 (python is 0 indexed) corresponds to the number of 100 intervals reached.

Finally, the output of the trained model, from the validation dataset can be found in ***vali_output.csv*** file located in the following directory: ***...vecmc_codes_zhonglin\rl\validation_data\***.

Sliding-window algorithm

This section describes the python implementation of the sliding-window algorithm used to solve optimization problems. Originally, this algorithm was used together to reduce the size of MILP problems involving thermal storages. As the MILP problem involving thermal storages is considerably large, the test-case included with the scripts attempt to solve the minimum hamiltonian path problem.

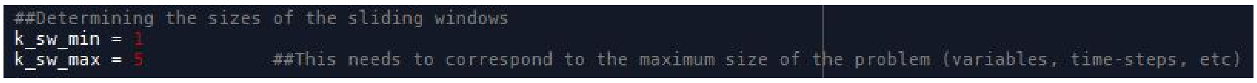
The codes needed to execute this algorithm can be found in the directory: ***...vecmc_codes_zhonglin\sw\***. The sample problem included in the folder, enables testing of the algorithm right off the bat. This can be done by simply executing the python file: ***...vecmc_codes_zhonglin\sw\main_script.py***. The results of the algorithm can be extracted from the folder: ***...vecmc_codes_zhonglin\sw\results\***.

1. Setting up the algorithm

Setting up this algorithm requires 2 files to be configured, the 1st allows the parameters of the sliding-window algorithm to be defined and the 2nd functions as a connector to the model to be solved/learned.

a. ***…\vecmc_codes_zhonglin\sw\main_script.py***

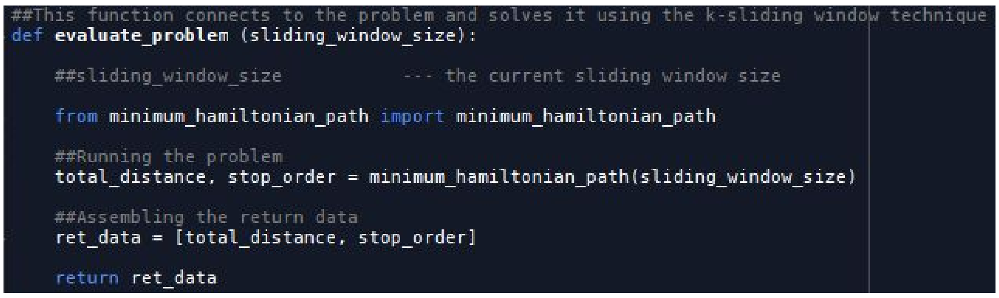
This file houses the main function (`run_k_sliding_window`) which defines the parameters and the flow of the algorithm. The following diagram illustrates the definition of the minimum and maximum size of the sliding-window to run the algorithm for.



```
##Determining the sizes of the sliding windows
k_sw_min = 1
k_sw_max = 5            ##This needs to correspond to the maximum size of the problem (variables, time-steps, etc)
```

b. ***…\vecmc_codes_zhonglin\sw\evaluate_problem.py***

This file houses the function which connects external models to the sliding-window algorithm. In the following example it is the minimum hamiltonian path problem (***minimum_hamiltonian_path.py***). The user should modify this function according to the problem to be solved.



```
##This function connects to the problem and solves it using the k-sliding window technique
def evaluate_problem (sliding_window_size):

    ##sliding_window_size           --- the current sliding window size

    from minimum_hamiltonian_path import minimum_hamiltonian_path

    ##Running the problem
    total_distance, stop_order = minimum_hamiltonian_path(sliding_window_size)

    ##Assembling the return data
    ret_data = [total_distance, stop_order]

    return ret_data
```

2. Running the algorithm

Once the parameters of the sliding-window algorithm and the associated problem to be solved has been defined, the only thing left to do is to run the algorithm. This is done by executing the file: ***.../vecmc_codes_zhonglin/sw/main_script.py*** in the spyder integrated development environment.

3. Extracting results

The results of the sliding-window algorithm can be found in ***results.csv*** file located in the folder: ***...vecmc_codes_zhonglin\sw\results\***. The first column indicates the sliding-window size, the second the optimal values of the variables and the last, the corresponding objective function.