Init functions explanation

Contents

[1. def simulation\_call(): 1](#_Toc129270599)

[2. fn\_output\_stat(activity, pop\_size=0): 1](#_Toc129270600)

[3. def fn\_output\_stat\_virtualcity(activity, pop\_size=0): 1](#_Toc129270601)

[4. Differences between fn\_output\_\_stat and fn\_output\_stat\_virtualcity functions 2](#_Toc129270602)

[5. def fn\_output\_od(activity): 2](#_Toc129270603)

[6. def fn\_output\_od\_mode\_balance(activity): 3](#_Toc129270604)

[7. Comparisson functions 3](#_Toc129270605)

[8. def fn\_quantify\_inadequacy\_stats(iter\_stats, city\_stats): 4](#_Toc129270606)

[9. def fn\_quantify\_inadequacy\_od(iter\_stats): 4](#_Toc129270607)

[10. def fn\_quantify\_inadequacy\_BCKUP(iter\_stats): 5](#_Toc129270608)

[11. def fn\_quantify\_inadequacy(iter\_stats): 5](#_Toc129270609)

[12. def fn\_process\_activities(): 5](#_Toc129270610)

[13. def fn\_simulation(params): 6](#_Toc129270611)

[14. def fn\_perform\_simulation(value, param\_def): 6](#_Toc129270612)

[**15.** **def fn\_perform\_simulation\_only(value, param\_def):** 6](#_Toc129270613)

[16. def fn\_scaleup\_standard(l\_sample, p\_def): 7](#_Toc129270614)

**def\_param\_update():**

# def simulation\_call():

This code imports the os library to interact with the operating system. The function simulation\_call checks whether MOCKING\_MODE is set to True, in which case it returns 0 without executing the simulation. If ACTIVITY\_FILE already exists, it is removed before the simulation is executed.

The sys\_call\_param variable is initialized as a list of two elements: the string "sh" and the value of SH\_EXEC\_SIMULATION. If CONTAINERIZED is set to True, the container home path and fingerprint are appended to the sys\_call\_param list.

The os.system() function is used to execute the shell command specified by sys\_call\_param. If ACTIVITY\_FILE exists after the simulation is executed, 0 is returned. Otherwise, an exception is raised with the value of the output variable.

# fn\_output\_stat(activity, pop\_size=0):

The Python function takes a Pandas DataFrame activity as input and returns a DataFrame containing output statistics. The function first checks if activity is empty and returns None if it is. It then calculates the number of people who travelled (pop\_travelled) by getting the length of the unique person\_id values in the activity DataFrame.

The activity DataFrame is then modified by adding a new column called od which is created by concatenating the prev\_stop\_location and stop\_location columns separated by an underscore.

Next, the function calculates output statistics in three different ways: total, purpose, and mode. The stat\_total DataFrame calculates the total number of tours and trips, as well as the average number of trips per tour for all activities. The stat\_purpose DataFrame calculates the same statistics for each tour type. The stat\_mode DataFrame calculates the same statistics for each stop mode. The agg function is used to aggregate the statistics for each group.

Finally, the three DataFrames (stat\_total, stat\_purpose, and stat\_mode) are combined into one DataFrame using the concat function. The resulting DataFrame is filtered to only the necessary columns.

# def fn\_output\_stat\_virtualcity(activity, pop\_size=0):

Inputs:

* activity : pandas DataFrame
* pop\_size :optional integer

Output:

* pandas DataFrame with statistics

The function first checks if the activity DataFrame is empty. If it is, the function returns None.

The next step is to compute the pop\_travelled variable, which is the number of unique person\_id in the activity DataFrame. The od column is then created by concatenating the prev\_stop\_location and stop\_location columns with an underscore in between. The stat\_total DataFrame is computed by grouping the activity DataFrame by person\_id and aggregating the number of unique tours (tours), unique stops (stops), and total trips (trips) for each person. The stat\_total DataFrame is then summed over all rows.

# Differences between fn\_output\_\_stat and fn\_output\_stat\_virtualcity functions

The main difference between the fn\_output\_stat and fn\_output\_stat\_virtualcity functions is that fn\_output\_stat\_virtualcity computes additional statistics related to the number of unique stop locations visited by each person and the population travel ratio (i.e., the ratio of unique persons who have travelled in the simulation to the total population size). These statistics are not computed in fn\_output\_stat.

In fn\_output\_stat\_virtualcity, the additional statistics are computed by first adding a new column od to the activity table that concatenates the previous stop location and the current stop location with an underscore in between. Then, the summarise function is used to compute the number of unique stops visited by each person (stops), which is used to compute the total number of stops visited by all people in the simulation. Similarly, the number of unique persons (travel\_ratio) who have visited each tourType or stop\_mode is computed and joined with the respective stat\_purpose and stat\_mode tables. Finally, the bind\_rows function is used to combine the various tables and compute statistics related to scope, type, variable, and value.

# def fn\_output\_od(activity):

The fn\_output\_od function takes an activity data frame as input and returns a matrix representing the Origin-Destination (O-D) travel counts between each pair of zones. The O-D matrix is a square matrix with row and column names representing the zones in the study area.

The function first extracts the list of zone names from the value column of the CITY\_STATS dictionary in the global setting\_obj object. It then creates a zero-filled square data frame with the zone names as row and column indices.

The function checks if the input activity data frame is empty and returns None if it is. Otherwise, it merges the activity data frame with the district\_map data frame in CITY\_STATS twice, once on prev\_stop\_location to get the origin zone code, and again on stop\_location to get the destination zone code.

The function then iterates over each row of the merged data frame and increments the corresponding element of the O-D matrix based on the origin and destination zone codes. Finally, the function returns the completed O-D matrix.

# def fn\_output\_od\_mode\_balance(activity):

This function takes as input a data frame called activity, which contains information about travel activity in a city. It calculates the origin-destination (O-D) matrix and the transportation mode balance for this activity, as well as the share of workers who are scheduled to take a trip.

The O-D matrix is a square matrix with rows and columns corresponding to different zones in the city, and with entries representing the number of trips taken between each pair of zones. This matrix is initialized with zeros using the emptyOD object from the global CITY\_STATS object, and then populated by iterating over the rows of activity and incrementing the appropriate entry in the matrix for each trip.

The transportation mode balance is a summary of the different modes of transportation used in the city, along with the estimated share of trips taken using each mode. This is calculated by grouping the rows of activity by mode category (e.g., public transit, car, bike, etc.), counting the number of trips in each group, and dividing by the total number of trips to get the estimated share. This summary is then joined with a table of target mode shares from the global CITY\_STATS object to get the final balance.

Finally, the share of workers scheduled to take a trip is calculated by filtering the rows of activity to include only trips associated with work (i.e., tourType == "Work") and selecting the primary stop for each trip (i.e., primary\_stop == T). The IDs of the workers associated with these trips are then extracted from the person\_id column and compared to a list of target worker IDs from the global CITY\_STATS object to get the share of workers who are scheduled to take a trip.

# Comparisson functions

The **fn\_quantify\_vector** function takes two vectors vector\_1 and vector\_2 and calculates the Euclidean distance between them using the NumPy sqrt and sum functions. It returns the resulting distance value.

The **fn\_quantify\_vector\_correlation** function also takes two vectors vector\_1 and vector\_2. Instead of calculating the distance between them, it calculates the Pearson correlation coefficient between the two vectors using the pearsonr function from the scipy.stats module. It returns the resulting correlation coefficient value. Note that the pearsonr function returns a tuple containing the correlation coefficient and the corresponding p-value. Therefore, we only return the correlation coefficient by accessing the first element of the tuple using the [0] index.

Both functions are relatively straightforward and can be used for comparing the similarity or dissimilarity between two vectors.

The **fn\_quantify\_matrix** function takes two matrices matrix\_1 and matrix\_2, and an optional mat\_weights matrix that specifies the weights to apply to each element of the matrices during the comparison. If mat\_weights is not provided, it is set to a matrix of ones with the same dimensions as matrix\_1.

The function calculates the absolute or squared difference between the two matrices depending on the values of the absolute and squared boolean arguments, respectively. Then, it returns the root-mean-square-error (RMSE) of the difference matrix **using the fn\_matrix\_rmse** function.

The fn\_matrix\_norm function calculates the Frobenius matrix norm of a matrix using the NumPy linalg.norm function.

The **fn\_matrix\_max\_cell\_loss** function calculates the maximum relative loss or error (loss per cell) between two matrices, given the mat and observed\_mat matrices. The function calculates the ratio of mat to observed\_mat and finds the number of cells with a relative error greater than 0.1. It then returns the maximum relative error plus the number of cells with a relative error greater than 0.1.The **fn\_matrix\_rmse** function calculates the RMSE of a matrix using the NumPy sqrt and mean functions. Finally, the **fn\_quantify** function is a wrapper function that calls **fn\_quantify\_matrix** with the same arguments and returns its result.

The **fn\_quantify\_inadequacy\_stats** function takes two arguments: iter\_stats, which is a pandas DataFrame containing statistics computed in a simulation iteration, and city\_stats, which is a pandas DataFrame containing observed statistics for a city.

# def fn\_quantify\_inadequacy\_stats(iter\_stats, city\_stats):

First, the function checks if **iter\_stats** is None. If it is, then it returns the maximum distance between the zero vector and city\_stats. This is because an empty activity schedule has been generated, and the distance between this schedule and the observed statistics should be as large as possible.

Next, the function merges iter\_stats with city\_stats based on the 'scope', 'type', and 'variable' columns using the pandas merge function. The resulting joined\_stats DataFrame has columns for scope, type, variable, value, and observed\_value.

The function then removes any rows in joined\_stats where observed\_value is NaN, which means that there is no observed value available for the corresponding scope, type, and variable combination.

Finally, the function calculates the distance between value and observed\_value for the remaining rows in joined\_stats using the fn\_quantify function, which presumably calculates some distance metric between two vectors. The resulting distance value is returned by the function.

# def fn\_quantify\_inadequacy\_od(iter\_stats):

The fn\_quantify\_inadequacy\_od function takes one argument iter\_stats which is a matrix containing travel distances between origins and destinations. The function first retrieves the setting\_obj object from the global environment. The object contains various settings related to the city statistics. If the object contains the weights attribute, it assigns the value to mat\_weights, otherwise, mat\_weights remains None. If iter\_stats is None, it means that no trips were generated, and the function returns the maximum distance by calling fn\_quantify with a matrix of zeros and setting\_obj["CITY\_STATS"]["value"]. If iter\_stats contains a matrix, the function calls fn\_quantify with iter\_stats and setting\_obj["CITY\_STATS"]["value"]. The mat\_weights are also passed to fn\_quantify if it is not None. Finally, the function returns the value returned by fn\_quantify.

# def fn\_quantify\_inadequacy\_BCKUP(iter\_stats):

The fn\_quantify\_inadequacy\_BCKUP function takes one argument iter\_stats, which is a dictionary containing various statistics related to the model output. The function first retrieves the setting\_obj object from the global environment. Then it calculates the od\_discrepancy by calling fn\_quantify\_inadequacy\_od function with iter\_stats["od"] as an argument.

Next, it sets balance\_discrepancy to 1. If the iter\_stats dictionary contains the balance key, it calls the fn\_quantify\_vector\_correlation function with the second and third columns of the balance array, and subtracts the result from 1 to get balance\_discrepancy.

Finally, the function calculates the value and base of the function, where the value is the product of balance\_discrepancy and the absolute difference between iter\_stats["total\_legs"] and setting\_obj["CITY\_STATS"]["total\_trips"]. The base is a list of od\_discrepancy, iter\_stats["total\_legs"], and balance\_discrepancy. The function returns a dictionary with the value and base as its keys.

The fn\_quantify\_inadequacy function takes in a dictionary (iter\_stats) containing information about a simulation iteration, such as the number of legs (total\_legs), the OD discrepancy (od), and the balance discrepancy (balance), and calculates an error value based on these inputs.

# def fn\_quantify\_inadequacy(iter\_stats):

In the first few lines of the function, the od\_discrepancy and balance\_discrepancy variables are initialized based on the input dictionary. If the input dictionary contains a workers\_work key, the workers\_work\_error variable is set to its value.

Next, the simulated and observed number of legs are obtained from the iter\_stats and setting\_obj dictionaries, respectively, and used to calculate the error\_val variable, which is a function of the od\_discrepancy, balance\_discrepancy, and workers\_work\_error.

Finally, the error\_val and an empty list (error\_base) are returned as a dictionary with keys 'value' and 'base', respectively. If a logger object exists in the shared.env dictionary and has a 'LOGGER' key, an informational message is logged using the log4r.info() function.

# def fn\_process\_activities():

* The function fn\_process\_activities() is defined in Python using the def keyword.
* The pandas library is used to read the CSV file into a dataframe.
* The to\_csv() method is used to save the dataframe as a CSV file.
* The output of the function is returned using the return keyword.

# def fn\_simulation(params):

* The function fn\_simulation() is defined in Python using the def keyword.
* The output of the function is a dictionary with keys "config\_output", "inadequacy", and "base\_residuals".
* The fn\_simulation\_config() function is called to update files with parameters and the output is assigned to param\_config\_output.
* The fn\_simulation\_call() function is called to execute the simulation.
* The fn\_process\_activities() function is called to read and process activity schedules and the output is assigned to outcome.
* The fn\_quantify\_inadequacy() function is called to quantify the (in)adequacy (discrepancy) of output statistics to the real world data (stats) and the output is assigned to inadequacy.
* If shared.env.settings.MOCKING\_MODE is True, then a random normal value is added to inadequacy["value"].
* The dictionary with the output is returned using the return keyword.

# def fn\_perform\_simulation(value, param\_def):

* The function fn\_perform\_simulation() takes in two arguments value and param\_def.
* The value argument is used to update the "value" column in the param\_def dataframe.
* The fn\_simulation() function is called with the updated param\_def dataframe to simulate the model and the output is assigned to param\_iter\_run.
* Warnings for particular parameter not being applied are left as TODO.
* The fn\_update\_value\_pools() function is called with value, param\_iter\_run["base\_residuals"], and param\_def as inputs to update the value pool.
* The function returns the inadequacy value from param\_iter\_run.

# def fn\_perform\_simulation\_only(value, param\_def):

* The function fn\_perform\_simulation\_only() takes in two arguments value and param\_def.
* The global settings object setting\_obj is obtained from shared.env.settings.
* The value argument is used to update the "value" column in the param\_def dataframe.
* The fn\_simulation\_config() function is called with param\_def as input to update the configuration files.
* The fn\_simulation\_call() function is called to execute the simulation.
* The activity schedule is loaded from a CSV file using pd.read\_csv() function and assigned to activity\_schedule.
* The activity schedule is saved to a CSV file with a filename based on the global settings object and the current time.
* The function returns the activity schedule as a Pandas DataFrame.

# def fn\_scaleup\_standard(l\_sample, p\_def):

The Python function first defines an inner function scale\_column that takes a column index c\_i and returns the scaled values for that column using the same formula as in the R function.

The np.apply\_along\_axis function is then used to apply scale\_column to each column of the sample matrix l\_sample. The first argument to np.apply\_along\_axis is the function to apply (scale\_column), the second argument specifies the axis along which to apply the function (0 for columns), and the third argument is an array of indices representing the columns to apply the function to (np.arange(l\_sample.shape[1])).

The scaled matrix is returned as the output of the function.

1. def fn\_preday\_sampling(vals=None, param\_space=None, sample\_size=1, \*\*kwargs):

The function fn\_preday\_sampling generates a sample for a set of parameters, where each parameter can be sampled using one of four sampling methods: tours, mode, od, and none. The function is quite long, and I will go over the code step-by-step to understand it better.

First, the function checks if the given vals and param\_space arguments are None or empty. If vals and param\_space['space'] is empty, then is\_initial is set to True. param\_space is a dictionary that contains information about the parameter space. The param\_space['definition'] contains information about the parameters, such as the parameter name, the initial value, the lower and upper limits, and the sampling method.

Next, the function gets the indexes of enabled parameters. An enabled parameter is a parameter that is not disabled in the parameter space. The function checks if the enabled field in param\_space['definition'] is True, and if it is, it gets the indexes of the enabled parameters.

The function then gets the names of the enabled parameters and the number of enabled parameters. It creates a generated sample numpy array that contains the initial values of the parameters.

If is\_initial is True, the function generates a new sample. It uses the lhs.create\_sample function to generate a Latin Hypercube Sample (LHS) for the enabled parameters. It stacks the initial values of the parameters and the generated sample and converts it to a structured numpy array with the same dtype as param\_space['definition'].

If the SAMPLING\_OPTIMAL\_TOLERANCE keyword argument is given and its value is greater than 0, the function finds the minimum value of the target column in the param\_space array and filters the rows where the target column value is less than the sum of the minimum value and SAMPLING\_OPTIMAL\_TOLERANCE. The resulting vals array contains the values of the enabled parameters that satisfy the filtering condition.

If there is only one row in the vals array, the function tiles it to create a new vals array with the same shape as the generated sample. Otherwise, it selects only the columns that correspond to the enabled parameters.

The function then sets the intensities for each sampling method. The intensity determines the probability of selecting a sampling method for a parameter. The higher the intensity, the higher the probability of selecting a sampling method.

The function then enables only the parameters in the active subspace. The active subspace is a subset of the parameter space that contains the parameters that are most relevant to the target variable.

The function sets up intensity values for each parameter based on the SAMPLING\_INTENSITY setting. If the residual pool for a sampling method is None or has less than two values, the function uses the first intensity value for that sampling method. Otherwise, the function computes the tail difference of the residual pool and compares it to a threshold. If the tail difference is less than the threshold for the focused sampling, the function uses the first intensity value. If the tail difference is less than the threshold for the spread sampling, the function uses the second intensity value. Otherwise, the function uses the first intensity value.

The function then generates a sample for each parameter. For each enabled parameter, the function gets the parameter definition, the intensity value for the sampling method, the size of the replaced sample, the current value of the parameter, the lower and upper bounds of the parameter, and the tunnel width of the parameter if it is specified.

After initializing the generated\_sample array with zeros, the code then loops through each enabled parameter using the enumerate function. For each parameter, it retrieves the parameter definition, intensity of its direct influence, and replaces a certain proportion of the current sample with new values sampled around the current value of the parameter.

If the parameter definition includes a tunnel\_width attribute, then the bounds for uniform sampling of new values are set to be p\_tunnel\_bound units away from the current value of the parameter. Otherwise, the bounds are set to the lower and upper limits of the parameter definition.

To sample new values, the code first creates an array of sample\_size copies of the current value of the parameter using np.repeat. It then selects a subset of indices to replace with new values using np.random.choice. The size of this subset is determined by the intensity of the parameter's direct influence, which is multiplied by sample\_size and rounded up to the nearest integer using int(np.ceil(...)).

Finally, the code samples new values from a normal distribution centered around the current value of the parameter, with standard deviation p\_tunnel\_bound, using np.random.normal. These new values are placed into the p\_sample array at the indices selected for replacement.

After generating the new parameter values for each enabled parameter, the code checks whether the initial parameter values have been imputed yet. If not, it checks whether the initial values for the enabled parameters match the values in the first row of the param\_space array. If not, it creates an array of the initial parameter values and appends it to the top of the generated\_sample array. Finally, it sets a flag in the shared\_env dictionary to indicate that the initial values have been imputed.

1. def update\_value\_pools(params, base\_residuals, param\_def):

The first parameter params is a list or array of parameter values that will be used to update the value pools. The second parameter base\_residuals is also a list or array of values that are used to compute the residuals of some system or model. The third parameter param\_def is not used in this function and it represents the parameters’ definitions. The function first retrieves a settings object from a global variable shared.env. If MOCKING\_MODE is set to True in this settings object, the function adds random noise to base\_residuals using the np.random.normal function.

The function then retrieves a dictionary called residual\_object from the param\_space dictionary stored in shared.env. The keys of residual\_object are components of the residual, such as "bias", "variance", or "noise". The function checks that the key "none" is not in residual\_object before proceeding.

The function then iterates over each key in residual\_components, which is the set difference between residual\_object.keys() and {'none'}. For each key comp, the function concatenates the corresponding residual component in residual\_object with the corresponding component in base\_residuals. The result is stored back into residual\_object.

1. def fn\_load\_params\_space\_definition(filepath, omit = [], sys\_modules = ["preday"], tunnel\_constraint = True, update\_initial = False):

This is a Python function called fn\_load\_params\_space\_definition that takes several arguments as input and returns a dictionary containing four elements: 'definition', 'space', 'value\_pool', and 'residual\_pool'.

The purpose of the function is to read in a parameter space definition file and create a parameter space for optimization. The file content is read in as a Pandas dataframe and filtered to include only rows that belong to the specified system modules and that have the 'include' column set to True. The function also extracts various columns from the dataframe and transforms them as necessary.

The omit argument is used to remove specific parameters from the definition. If omit is a string, the function assumes that it is a boolean column in the dataframe and extracts the list of parameters that have a True value for that column. If omit is a list, the function removes all parameters whose 'parameter' column is in the list.

The tunnel\_constraint argument is a boolean that determines whether or not to apply tunnel constraints to the parameter space. If True, the function sets the lower and upper limits for each parameter based on its current value and a predefined tunnel width. The width is specified as 2.0 for most parameters, except for those with 'logsum' in their name, which have a restricted width of 0.05.

Finally, the function returns a dictionary with the following elements:

* definition: a dataframe containing the filtered and transformed file content, with additional columns for initial value, changed status, and initial lower and upper limits.
* space: a transposed dataframe containing the initial parameter values, with parameter names as column headers.
* value\_pool: a dictionary with parameter names as keys and empty lists as values, used for storing parameter values during optimization.
* residual\_pool: a dictionary with keys for 'tours', 'mode', and 'od', used for storing various residuals during optimization.