
MARIO

Release 0.1

Mohammad Amin Tahavori, Lorenzo Rinaldi, Nicolo Golinucci

Nov 25, 2022

CONTENTS

1	MARIO	3
1.1	What is it	3
1.2	Requirements	4
1.3	Recommended installation method	4
1.4	Quickstart	4
1.5	Python module requirements	5
1.6	License	6
2	Download and installation	7
2.1	Requirements	7
2.2	Recommended installation method	7
3	Contributors	9
3.1	Communication, issues, bugs and contributions	9
4	Terminology	11
5	Handling databases	13
5.1	Exiobase3 parser	13
5.2	EORA parser	13
5.3	Multi regional monetary supply and use exiobase	13
5.4	Parsing using pd.DataFrame	13
5.5	Parsing EUROSTAT	13
5.6	Excel parser	13
5.7	Plotting by plot_matrix	13
6	MARIO example gallery	15
6.1	Add Extensions	15
6.2	Shock and scenario analysis	15
7	API Reference	17
7.1	Analyzing database	17
7.2	Database modification	20
7.3	Shock analysis	25
7.4	Data visualization	26
7.5	Get excels	29
7.6	Save data	31
7.7	Database parsers	32
7.8	Calculations	36
7.9	Metadata	45
7.10	Test	45

7.11	Directory	46
7.12	Utilities	46
7.13	Logging	47
Index		49

Multifunctional Assessment of Regions through Input-Output

Developed by [SESAM Group](#), Department of Energy, Politecnico di Milano

Contents:



Multifunctional Analysis of Regions through Input-Output. ([Documents](#))

1.1 What is it

MARIO is a python package for handling input-output tables and models inspired by [pymrio](#). MARIO aims to provide a *simple & intuitive* API for common IO tasks without needing in-depth programming knowledge. MARIO supportst automatic parsing of different structured tables such EXIOBASE, EORA, EUROSTAT in different formats namely:

- Single region
- Multi region
- Hybrid tables
- Monetary tables
- Input-Output tables
- Supply-Use tables

When databases are not structured, MARIO supports parsing data from xlsx, csv, txt files or pandas.DataFrames.

More than parsing data, MARIO includes some basic functionalities:

- Aggregation of databases
- SUT to IOT transformation
- **Modifying database in terms of adding:**
 - New sectors, activities or commodities to the database
 - Adding new extensions to the satellite account
- Scneario and shock analysis
- Backward and forward linkages analysis
- Extracting single region database from multi region databases
- Balance test
- Productivity test
- Exporting the databases into different formats for scenarios analyzed
- Interactive visualization routines

1.2 Requirements

MARIO has been tested on macOS and Windows.

To run MARIO, a couple of things are needed:

1. Being in love with Input-Output :-)
2. The Python programming language, version 3.7 or higher
3. A number of Python adds-on packages
4. For some functionalities a solver may needed (optional)
5. MARIO software itself

1.3 Recommended installation method

The easiest way to make MARIO software working is to use the free conda package manager which can install the current and future MARIO dependencies in an easy and user friendly way.

To get conda, [download and install “Anaconda Distribution”](#) . Between different options for running python codes, we strongly suggest, [Spyder](#), which is a free and open source scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts.

You can install mario using pip or from source code. It is suggested to create a new environment by running the following command in the anaconda prompt

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

If you create a new environment for mario, to use it, you need to activate the mario environment each time by writing the following line in *Anaconda Prompt*

```
conda activate mario
```

Now you can use pip to install mario on your environment as follow:

```
pip install mariopy
```

You can also install from the source code!

1.4 Quickstart

A simple test for Input-Output Table (IOT) and Supply-Use Table (SUT) is included in mario.

To use the IOT test, call

```
import mario
test_iot = mario.load_test('IOT')
```

and to use the SUT test, call

```
test_sut = mario.load_test('SUT')
```

To see the configurations of the data, you can print them:


```
print(test_iot)
print(test_sut)
```

To see specific sets of the tables like regions or value added, `get_index` function can be used:

```
print(test_iot.get_index('Region'))
print(test_sut.get_index('Factor of production'))
```

To visualize some data, various plot functions can be used:

```
test_iot.plot_matrix(...)
```

Specific modifications on the database can be done, such as SUT to IOT transformation:

```
reformed_iot = test.sut_to_iot(method='B')
```

The changes can be tracked by metadata. The history can be checked by calling:

```
reformed_iot.meta_history
```

The new database can be saved into excel,txt or csv file:

```
reformed_iot.to_excel(path='a folder//database.xlsx')
```

1.5 Python module requirements

Some of the key packages the mario relies on are:

- Pandas
- Numpy
- Plotly
- Tabulate
- Cvxpy (Optional in this version)

Note:

- This project is under active development.
 - More examples will be uploaded through time to the gellery.
 - More parsers will be added to the next version.
 - The next version will cover some optimization models within the IO framework
-

1.6 License



This work is licensed under a [Attribution-NonCommercial-ShareAlike 4.0 International \(CC BY-NC-SA 4.0\)](https://creativecommons.org/licenses/by-nc-sa/4.0/)

DOWNLOAD AND INSTALLATION

2.1 Requirements

MARIO has been tested on macOS and Windows.

To run MARIO, a couple of things are needed:

1. Being in love with Input-Output :-)
2. The Python programming language, version 3.7 or higher
3. A number of Python adds-on packages
4. For some functionalities a solver may needed (optional)
5. MARIO software itself

2.2 Recommended installation method

The easiest way to make MARIO software working is to use the free conda package manager which can install the current and future MARIO dependencies in an easy and user friendly way.

To get conda, [download and install “Anaconda Distribution”](#) . Between different options for running python codes, we strongly suggest, [Spyder](#), which is a free and open source scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts.

You can install mario using pip or from source code. It is suggested to create a new environment by running the following command in the anaconda prompt

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

If you create a new environment for mario, to use it, you need to activate the mario environment each time by writing the following line in *Anaconda Prompt*

```
conda activate mario
```

Now you can use pip to install mario on your environment as follow:

```
pip install mariopy
```

You can also install from the source code!

CONTRIBUTORS

mario has been developed within [SESAM group](#) in the Department of Energy Engineering at Politecnico di Milano. The research activity of SESAM focuses on the use of mathematical models for the study of systems and components in the energy field and industrial ecology.

mario has been used for some research projects within the group, and now is available as an open source code for the Input-Output modelling communities.

- [Mohammad Amin Tahavori](#) : Code development & functional design
- [Lorenzo Rinaldi](#) : Code development & functional design
- [Nicolò Golinucci](#) : Code development & functional design
- [Emanuela Colombo](#) : Scientific coordination
- [Matteo Vincenzo Rocco](#) : Scientific coordination

3.1 Communication, issues, bugs and contributions

We use github for tracking bugs, issues and suggestions related to mario. Any communications thorough email are welcomed.

You may also follow us on social media like twitter or our official website to follow last news on mario or our research. We are looking forward for future communications and contributions.

TERMINOLOGY

In the lack of consistent terminology for IO systems in the scientific community, MARIO uses its own customized variable names. MARIO follows a thermodynamic way of nomenclature which:

- Uppercase letters represents Flows
- Lowercase letters represents Coefficients

Following table represents the variables and their explanations in MARIO:

Table 1: MARIO Terminology

variable name	also known as	extended name
Z	T	Intersectoral transaction flows matrix
z	A	Intersectoral transaction coefficients matrix
w	L	Leontief coefficient matrix
Y	F	Final demand matrix
X	x, q, g	Production vector
V	F	Factor of production transaction flows matrix
v	f, B, S	Factor of production transaction coefficients matrix
E	F, D_pba, terr	Satellite transaction flows matrix
U	T	Use transaction flow matrix
u	A	Use coefficients matrix
S	V, M, T	Supply transaction flow matrix
s	A	Supply coefficients matrix
EY	S_Y, F_hh, F_y	Satellite transaction flows matrix for final use
M	...	Economic impact matrix
m	M	Multipliers coefficient matrix
F	D_cba, con	Footprint matrix
e	f, B, S	Satellite transaction coefficients matrix
f	M	Footprint coefficients matrix
g	G	Gosh coefficients matrix
b	B	Intersectoral transaction direct-output coefficients matrix
p	...	Price index coefficients vector

HANDLING DATABASES

5.1 Exiobase3 parser

5.2 EORA parser

5.3 Multi regional monetary supply and use exiobase

5.4 Parsing using pd.DataFrame

5.5 Parsing EUROSTAT

5.6 Excel parser

5.7 Plotting by plot_matrix

MARIO EXAMPLE GALLERY

6.1 Add Extensions

6.2 Shock and scenario analysis

API REFERENCE

7.1 Analyzing database

<code>CoreModel.is_balanced(method[, data_set, ...])</code>	Checks if a specific data_set in the database is balance or not
<code>CoreModel.is_productive(method[, data_set])</code>	Checks the productivity of the system
<code>CoreModel.is_multi_region</code>	Defines if a database is single region or multi-region
<code>CoreModel.is_hybrid</code>	checks if the database is hybrid or monetary
<code>CoreModel.sets</code>	Returns a list of levels of info in the model
<code>CoreModel.scenarios</code>	Returns all the scenarios existed in the model
<code>CoreModel.table_type</code>	Returns the type of the database
<code>CoreModel.get_index(index[, level])</code>	Returns a list or a DataFrame of different levels of indices in the database.

7.1.1 mario.CoreModel.is_balanced

`CoreModel.is_balanced(method, data_set='baseline', margin=0.05, as_dataframe=False)`

Checks if a specific data_set in the database is balance or not

Note: If the dataset is not balance, a table will be printed spotting the imbalances.

Parameters

- **method** (*str*) – represents the method to check the balance:
 1. 'flow'
 2. 'coefficient': (zeros won't be considered)
 3. 'price': (zeros won't be considered)
- **data_set** (*str*) – defines the scenario to be checked
- **margin** (*float*) – float which will be considered as a margin for the balance
- **as_dataframe** (*boolean*) – if True, in case that database is not balance, will return a `pd.DataFrame` spotting the imbalances

Returns

- *if balance* – returns True

- *if not balance* – if `as_dataframe= False`, return a boolean (False) if `as_dataframe= True`, return a `pd.DataFrame`

7.1.2 `mario.CoreModel.is_productive`

`CoreModel.is_productive(method: str, data_set: str = 'baseline') → bool`

Checks the productivity of the system

Parameters

- **method** (*str*) – represents the method to check the balance:
 1. 'flow'
 2. 'coefficient'
 3. 'price'
- **data_set** (*str*) – defining the scenario to be checked
- **margin** (*float which will be considered as a margin for the balance*) –

Returns

True if the dataset is balance False if the dataset is not balance → it also prints in a table the imbalances

Return type

boolean

7.1.3 `mario.CoreModel.is_multi_region`

property `CoreModel.is_multi_region`

Defines if a database is single region or multi-region

Returns

True if database is multi region else False

Return type

boolean

7.1.4 `mario.CoreModel.is_hybrid`

property `CoreModel.is_hybrid`

checks if the database is hybrid or monetary

Note:

- for IOT table checks the unity of the Secotr + Factor of production
 - for SUT table checks the unity of the Activity+Commodity+Factor of production
-

Return type

bool

7.1.5 mario.CoreModel.sets

property `CoreModel.sets`

Returns a list of levels of info in the model

Return type

lists

7.1.6 mario.CoreModel.scenarios

property `CoreModel.scenarios`

Returns all the scenarios existed in the model

Return type

list

7.1.7 mario.CoreModel.table_type

property `CoreModel.table_type`

Returns the type of the database

Return type

str

7.1.8 mario.CoreModel.get_index

`CoreModel.get_index(index, level='main')`

Returns a list or a DataFrame of different levels of indeces in the database.

Parameters

- **index** (*str*) – if ‘all’ return all the indeces else representing the level such as Region, Sector, ...
- **level** (*str*) – main for the main indeces and aggregated for aggregated level if exists

Returns

- *dict* – if index=‘all’ returns a dictionary of all indeces
- *list* – if index is not ‘all’ returns a list of requested index level

7.2 Database modification

<code>Database.aggregate(io[, drop, levels, ...])</code>	This function is in charge of reading data regarding the aggregation
<code>Database.add_sectors(io, new_sectors, ..., ...)</code>	Adds a Sector/Activity/Commodity to the database
<code>Database.to_single_region(region[, backup, ...])</code>	Extracts a single region from multi-region databases
<code>Database.sut_to_iot(method[, inplace])</code>	The function will transform a SUT table to a IOT table
<code>Database.add_extensions(io, matrix, units[, ...])</code>	Adding a new extension [Factor of production or Satellite account] to the database passing the coefficients or the absolute values.
<code>Database.update_scenarios(scenario, **matrices)</code>	Updates the matrices for a specific scenario.
<code>Database.reset_to_flows(scenario[, backup])</code>	Deletes the coefficients of a scenario and keeps only flows
<code>Database.reset_to_coefficients(scenario[, ...])</code>	Deletes the flows of a scenario and keeps only coefficients
<code>Database.clone_scenario(scenario, name)</code>	Creates a new scenario by cloning an existing scenario
<code>Database.copy()</code>	Returns a deepcopy of the instance
<code>Database.backup()</code>	The function creates a backup of the last configuration of database to be returned in case needed.
<code>Database.reset_to_backup()</code>	This function is in charge of resetting back the database to the last back-up.

7.2.1 mario.Database.aggregate

`Database.aggregate(io, drop=['unused'], levels='all', backup=True, calc_all=True, ignore_nan=False, inplace=True)`

This function is in charge of reading data regarding the aggregation
of different levels and aggregate data.

Parameters

- **io** (*str*, *Dict*[*pd.DataFrame*]) –
 1. in case that the data should be given through an Excel file, represents the path of the Excel file
 2. in case that the data needs to be given by DataFrame, a dictionary of DataFrames can be give in which the keys are the name of the levels and values are the DataFrames
- **drop** (*str*, *List*[*str*]) – representing the items/items that should be dropped (only allowed for E and EY matrix)
- **levels** (*str*, *List*[*str*]) –
 1. in case that a single level or 'all' levels should be aggregated, str can be used
 2. in case that multiple levels should be aggregated, a list of levels should be used
- **backup** (*boolean*) – shows if the user needs to make a backup before aggregating so can reset to previous case.
- **calc_all** (*boolean*) – if True, ['v','z','e'] will be calculated automatically after the aggregation of flows
- **ignore_nan** (*boolean*) –

1. if False, will stop the code if finds some nan values in the aggregation dataframes
 2. if True, will ignore the NaNs and do not aggregated the specific items with NaN values
- **inplace** (*boolean*) – if True will aggregate the database object itself otherwise will return the aggregated object as a new database object

Returns

- *mario.Database* – if inplace = False returns a new mario.Database
- *None* – if inplace = True implements the changes in place

7.2.2 mario.Database.add_sectors

`Database.add_sectors(io, new_sectors, regions, item, inplace=True, notes=None)`

Adds a Sector/Activity/Commodity to the database

Note: This function will delete all the scenarios in the database and overwrite the new matrices to the baseline.

Parameters

- **io** (*str*, *Dict[pd.DataFrame]*) – the path of the Excel file containing the information or an equal dictionary with keys as the names of the sheets and values as dataframes of the excel file
- **new_sectors** (*list*) – new sectors/activities/commodities to be added to the database
- **regions** (*list*) – specific regions that the new technology will be specified
- **item** (*str*) – the item to be added. Sector for IOT table and Activity or Commodity for SUT Sector if IOT, Activity or Commodity if SUT
- **inplace** (*boolean*) – if True will implement the changes directly in the database else returns a new new mario.Database
- **notes** (*list*, *Optional*) – notes to be recorded in the metadata

Returns

- *mario.Database* – if inplace = True will return a new mario.Database
- *None* – if inplace = False returns None and implements the changes in the databasaes

7.2.3 mario.Database.to_single_region

`Database.to_single_region(region, backup=True, inplace=True)`

Extracts a single region from multi-region databases

Note: Following assumptions are considered (on flow matrices):

- intermediate imports accounted as 'Import' in V
- intermediate exports are accounted as 'Intermediate exports' in Y
- final demand exports are accounted as 'Final demand exports' in Y
- EY is accounted only for local final demand

Parameters

- **region** (*str*) – the region to extract
- **backup** (*boolean*) – if True, creates a backup of the database before changes
- **inplace** (*boolean*) – if True, changes the database inplace otherwise, returns a new object

Returns

- *mario.Database* – if *inplace*= True returns a new *mario.Databases*
- *None* – if *inplace*= False, changes the database inplace

7.2.4 `mario.Database.sut_to_iot`

`Database.sut_to_iot(method, inplace=True)`

The function will transform a SUT table to a IOT table

Note: Calling this function will delete all the existing scenarios in the database and create the new baseline scenario.

Parameters

- **method** (*str*) – Defines the method for transformation of the database:
 - A. Product-by-product input-output table based on product technology assumption (possible negative values)
 - B. Product-by-product input-output table based on industry technology assumption
 - C. Industry-by-industry input-output table based on fixed industry sales structure assumption (possible negative values)
 - D. Industry-by-industry input-output table based on fixed product sales structure assumption
- **inplace** (*boolean*) – if True, implements the changes on the Database else returns a new object without changing the original Database object

Returns

- *None* – if *inplace* True
- *mario.Database* – if *inplace* False

7.2.5 `mario.Database.add_extensions`

`Database.add_extensions(io, matrix, units, backup=True, inplace=True, calc_all=True, notes=None, EY=None)`

Adding a new extension [Factor of production or Satellite account] to the database passing the coefficients or the absolute values.

Note: This function will delete all the existing scenarios and implement the new sets of the matrices in the baseline.

Parameters

- **io** (*str*, *pd.DataFrame*) – if the data is given from an excel file, is the path to the file else is a *pd.DataFrame*
- **matrix** (*str*) – defines where the new extension should be added to. The options are :
 - 'v' value added by coefficient
 - 'V' value added by absolute value
 - 'e' satellite account by coefficient
 - 'E' satellite account by absolute
- **units** (*pd.DataFrame*) – a dataframe whose rows are the items to be added and single column which contains the units for every row
- **backup** (*boolean*) – creates a backup of the last database to recover if needed
- **inplace** (*boolean*) – if True, will change the database inplace otherwise will return a new object
- **calc_all** (*boolean*) – if True, will calculate the main missing data
- **notes** (*list*, *Optional*) – to add notes to the metadata
- **EY** (*pd.DataFrame*, *Optional*) – In case that E,e are used as the matrix, EY can be updated too

Returns

- *mario.Database* – if inplace= True returns a new *mario.Databases*
- *None* – if inplace= False, changes the database inplace

7.2.6 mario.Database.update_scenarios

`Database.update_scenarios(scenario, **matrices)`

Updates the matrices for a specific scenario.

Note: using update scenarios, will update only the matrices passed. In case, that the update, impacts other matrices, this should be done manually using update_scenarios and updating other matrices or resetting the databases using reset_to_flows or reset_to_coefficients and recalculate the matrices based on the inputs.

Parameters

- **scenario** (*str*) – the name of the scenario
- **matrices** (*pd.DataFrame*) – dict of the matrices as dataframes (keys are the name of the matrices and values are the DataFrames)

Example

To update the z and v matrices in example object for scenario baseline with new_z and new_v

```
example.update_scenarios(scenario='baseline', z=new_z, v=new_v)
```

7.2.7 mario.Database.reset_to_flows

`Database.reset_to_flows(scenario, backup=True)`

Deletes the coefficients of a scenario and keeps only flows

Parameters

- **scenario** (*str*) – the specific scenario to reset
- **backup** (*boolean*) – if True, will create a backup of database before changes

7.2.8 mario.Database.reset_to_coefficients

`Database.reset_to_coefficients(scenario, backup=True)`

Deletes the flows of a scenario and keeps only coefficients

Parameters

- **scenario** (*str*) – the specific scenario to reset
- **backup** (*boolean*) – if True, will create a backup of database before changes

7.2.9 mario.Database.clone_scenario

`Database.clone_scenario(scenario, name)`

Creates a new scenario by cloning an existing scenario

Parameters

- **scenario** (*str*) – from which scenario clone
- **name** (*str*) – the name of the new scenario to be created

Example

Creating a new scenario called scenario_2 by cloning the data in scenario baseline

```
database.clone_scenario(scenario='baseline', name='scenario_2')
```

7.2.10 mario.Database.copy

Database.**copy**()

Returns a deepcopy of the instance

Return type

MARIO.Database

7.2.11 mario.Database.backup

Database.**backup**()

The function creates a backup of the last configuration of database to be returned in case needed.

7.2.12 mario.Database.reset_to_backup

Database.**reset_to_backup**()

This function is in charge of resetting back the database to the last back-up. All the matrices and indices will be updated to the last back-up

7.3 Shock analysis

Database.shock_calc(io[, z, e, v, Y, notes, ...])

Implements shocks on different matrices with the possibility of defining clusters on every level of information.

7.3.1 mario.Database.shock_calc

Database.**shock_calc**(io, z=False, e=False, v=False, Y=False, notes=[], scenario=None, force_rewrite=False, **clusters)

Implements shocks on different matrices with the possibility of defining clusters on every level of information.

Note:

- Shocks can be implemented only with respect to the baseline
 - Shocks will be implemented only on coefficients
-

Parameters

- **io** (*str*, *Dict[pd.DataFrame]*) – pass a str defining the excel file containing the shock data or pass a dict of dataframes in which keys are the name of matrices and values are the dataframes of the shock (exactly the same format of excel file)
- **z** (*boolean*) – if True will implement shock on the Z or z
- **e** (*boolean*) – if True will implement shock on the E or e
- **v** (*boolean*) – if True will implement shock on the V or v
- **Y** (*boolean*) – if True will implement shock on Y

- **notes** (*list*) – extra info can be recorded in the metadata
- **scenario** (*str*, *Optional*) – the name for the scenario implemented, in the instance.matrices. If nothing passed, default names will be considered (shock #)
- **fore_rewrite** (*boolean*) – if False will avoid overwriting existing scenario
- ****cluster** (*dict*) – can be used to implement complex shocks by defining clusters (refer to tutorials)

7.4 Data visualization

<code>mario.set_palette([mario_palettes, user_palette])</code>	Sets the default palette of plots
<code>Database.plot_gdp([path, plot, scenario, ...])</code>	Plots sectoral GDP with additional info
<code>Database.plot_bubble(x, y, size[, path, ...])</code>	Creates bubble plots
<code>Database.plot_linkages([scenarios, ...])</code>	Plots linkages in different modes
<code>Database.plot_matrix(matrix, x, color[, y, ...])</code>	Generates a general html barplot giving the user certain degrees of freedom such as:

7.4.1 mario.set_palette

`mario.set_palette(mario_palettes=None, user_palette=None)`

Sets the default palette of plots

Note: if enough colors are not assigned to the palette non-duplicate random colors will be added to palette when needed

Parameters

- **mario_palettes** (*str*) – choosing between mario default palettes
- **user_palette** (*list*) – a list of user palettes

7.4.2 mario.Database.plot_gdp

`Database.plot_gdp(path=None, plot='treemap', scenario='baseline', extension=None, extension_value='relative', auto_open=True, drop_reg=None, title=None)`

Plots sectoral GDP with additional info

Parameters

- **path** (*str*, *Optional*) – the path and the name of the file to save the plot
- **plot** (*str*) – type of the plot ['treemap', 'sunburst']
- **scenario** (*str*) – scenario to plot
- **extension** (*str*, *optional*) – a satellite account item that can be used for scaling the colors
- **extension_value** (*str*) – # 'relative' for scaling on specific satellite account (e.g. CO2/Euro of production) # 'absolute' for absolute scaling on satellite account (e.g. total CO2)

- **auto_open** (*boolean*) – if True, the plot will be opened automatically
- **drop_reg** (*str, optional*) – a region to be excluded in the plot can be passed. Useful when using MRIO with one region and a Rest of the World region.
- **title** (*str, optional*) – here the user can pass a costume title for the plot

7.4.3 mario.Database.plot_bubble

`Database.plot_bubble(x, y, size, path=None, auto_open=True, scenario='baseline', log_x=False, log_y=False)`

Creates bubble plots

Parameters

- **x** (*str*) – item to locate on x-axis. valid items should be a factor of production, satellite account or GDP
- **y** (*str*) – item to locate on y-axis. valid items should be a factor of production, satellite account or GDP
- **size** (*str*) – item to locate on size of bubble. valid items should be a factor of production, satellite account or GDP
- **path** (*str*) – the path and the name of the file to save the plot. Like 'pathplot.html'
- **auto_open** (*boolean*) – if True, opens the plot automatically
- **scenario** (*str*) – scenario to plot
- **log_x** (*boolean*) – if True, will plot with x-axis with Logarithmic scale
- **log_y** (*boolean*) – if True, will plot with y-axis with Logarithmic scale

7.4.4 mario.Database.plot_linkages

`Database.plot_linkages(scenarios='baseline', normalized=True, cut_diag=True, multi_mode=False, path=None, plot='Total', auto_open=True, **config)`

Plots linkages in different modes

Note: when caclulating linkages, possible negative numbers, are ignore

Parameters

- **scenarios** (*str, List[str]*) – A scenario or a list of scenarios to plot
- **normalized** (*boolean*) – if True, plots normalized linkages
- **cut_diag** (*boolean*) – if True, ignores the self consumption of sectors in calculating linkages
- **multi_mode** (*boolean*) – ~~TODO~~
- **path** (*str, Optional*) – the path and the name of the plot file. (path should contain the name of the file with .html extension) for example 'pathlinkagesPlot.html'
- **plot** (*str*) – Options are:
 - 'Total' to plot the total linkages

- 'Direct' to plot the direct linkages
- **auto_open** (*boolean*) – if True, opens the plot automatically

7.4.5 mario.Database.plot_matrix

Database.plot_matrix(*matrix, x, color, y='Value', item='Sector', facet_row=None, facet_col=None, animation_frame='Scenario', base_scenario=None, path=None, mode='stack', layout=None, auto_open=True, shared_yaxes='all', shared_xaxes=True, **filters*)

Generates a general html barplot giving the user certain degrees of freedom such as:

- Regions (both the ones on the indices and columns)
- Sectors/Commodities/Activities (both the ones on the indices and columns)
- Scenarios
- Units

Parameters

- **matrix** (*str*) – Matrix to be plotted. Three families of matrix can be read according to their intrinsic structure:
 1. The first family includes only matrix 'X', which has 3 levels of indices and 1 level of columns
 2. The second family includes matrices 'Z','z','U','u','S','s','Y', which have 3 levels of indices and 3 levels of columns
 3. The third family includes matrices 'E','e','V','v','EY', which have 1 level of indices and 3 levels of columns
- **path** (*str*) – Path where to save the html file
- **x** (*str*) – Degree of freedom to be showed on the x axis. Acceptable options change according to the matrix family
- **y** (*str*) – Degree of freedom to be showed on the y axis. Default y='Value'. Acceptable options change according to the matrix family
- **item** (*str*) – Indicates the main level to be plot. Possible options are "Commodity","Activity" for SUT tables and "Sector" for IOT tables. It is mandatory to be defined only for SUT tables. For "Z","z","U","u","S","s","Y","X", it selects the rows level between 'Activity' and 'Commodity'. For "V","v","E","e","EY","M","F", it selects the columns level between 'Activity' and 'Commodity'.
- **facet_row** – String referring to one level of indices of the given matrix. Values from this column or array_like are used to assign marks to faceted subplots in the vertical direction
- **facet_col** – String referring to one level of indices of the given matrix. Values from this column or array_like are used to assign marks to faceted subplots in the horizontal direction
- **animation_frame** – Defines whether to switch from one scenario to the others by means of sliders
- **base_scenario** (*str*) – By setting None, the passed matrix will be displayed for each scenario available. By setting this parameter equal to one of the scenarios available, the passed matrix will be displayed in terms of difference with respect to each of the other scenarios. In this last case, the selected scenario will not be displayed

- **mode** (*str*) – Equivalent to `plotly.grap_object.figure.update_layout bargmode`. Determines how bars at the same location coordinate are displayed on the graph. * With “stack”, the bars are stacked on top of one another * With “relative”, the bars are stacked on top of one another * With “group”, the bars are plotted next to one another centered around the shared location. * With “overlay”, the bars are plotted over one another, you might need to an “opacity” to see multiple bars.
- **auto_open** (*boolean*) – if True, it opens automatically the saved file in the default html reader application
- **filters** (*dict*) – The user has the option to filter the sets according to the necessity. Acceptable options are the following and must be provided as list:
 - ‘filter_Region_from’,
 - ‘filter_Region_to’,
 - ‘filter_Sector_from’,
 - ‘filter_Sector_to’,
 - ‘filter_Consumption category’,
 - ‘filter_Activity’,
 - ‘filter_Commodity’

7.5 Get excels

mario has some functions providing automatic excel file generations for easing some of the functionalities such as aggrgeaton and adding sectors.

<code>Database.get_aggregation_excel([path, levels])</code>	Generates the Excel file for aggregation of the database
<code>Database.get_extensions_excel(matrix[, path])</code>	Generates an Excel file for easing the add extension functionality
<code>Database.get_add_sectors_excel(new_sectors, ...)</code>	Generates an Excel file to add a sector/activity/commodity to the database
<code>Database.get_shock_excel([path, num_shock])</code>	Creates an Excel file based on the shape and the format

7.5.1 mario.Database.get_aggregation_excel

`Database.get_aggregation_excel(path=None, levels='all')`

Generates the Excel file for aggregation of the database

Parameters

- **path** (*str*) – path to generate an Excel file for database aggregation. The Excel file has different sheets named by the level to be aggregated.
- **levels** (*str*) – levels to be printed as Excel sheets. If ‘all’ it will print out all the levels, else different levels should be passed as a list of levels such as [‘Region’, ‘Sector’]

7.5.2 mario.Database.get_extensions_excel

Database.get_extensions_excel(*matrix*, *path=None*)

Generates an Excel file for easing the add extension functionality

Parameters

- **matrix** (*str*) – the name of the matrix to add extensions ['V','E']
- **path** (*str*) – defines the of the Excel file to save the Excel file such as: Extensions.xlsx

7.5.3 mario.Database.get_add_sectors_excel

Database.get_add_sectors_excel(*new_sectors*, *regions*, *path=None*, *item=None*)

Generates an Excel file to add a sector/activity/commodity to the database

Parameters

- **new_sectors** (*list*) – new sectors/activities/commodities to be added to the database
- **regions** (*list*) – specific regions that the new technology will be specified
- **path** (*str*) – the path in which the Excel file will be saved (path should contain the name of file like 'pathadd_sector.xlsx')
- **item** (*str*) – the item to be added. Sector for IOT table and Activity or Commodity for SUT

7.5.4 mario.Database.get_shock_excel

Database.get_shock_excel(*path=None*, *num_shock=10*, ***clusters*)

Creates an Excel file based on the shape and the format
of the database for the shock implemntation.

Note: The generated Excel file will have list validations to simplify the error handling and help the user. In case the number of shocks are more than 10, it is suggested to increse num_shock to have more validated rows in every sheet of the Excel file.

Parameters

- **path** (*str*) – defines the path which the Excel file will be stored
- **clusters** (*dict*) – nested dictwith clusters the user can define a sets of clusters for more specificed shock implementation. e.g. clusters = {'Region':{'cluster_1':['reg1','reg2']}}

7.6 Save data

<code>Database.to_excel([path, flows, ...])</code>	Saves the database into an Excel file
<code>Database.to_txt([path, flows, coefficients, ...])</code>	Saves the database multiple text file based on given inputs

7.6.1 mario.Database.to_excel

`Database.to_excel(path=None, flows=True, coefficients=False, units=True, scenario='baseline')`

Saves the database into an Excel file

Note:

- The function will create a single Excel file with different sheets.
 - The sheets based on the inputs will be:
 - coefficients
 - flows
 - units
 - It is suggested to keep the `units = True` so the output file can be used to parse with MARIO again.
-

Parameters

- **path** (*str*) – the path that the Excel file should be saved. If it is `None`, MARIO will try to use the default path and inform the user with a warning. (the path should contain the name of excel file like 'pathdatabase.xlsx')
- **flows** (*boolean*) – if `True`, in the Excel file, a sheet will be created named flows containing the data of the flows
- **coefficients** (*boolean*) – if `True`, in the Excel file, a sheet will be created named coefficients containing the data of the coefficients
- **units** (*boolean*) – if `True`, in the Excel file, a sheet will be created named units containing the data of the units
- **scenario** (*str*) – defines the scenario to print out the data

7.6.2 mario.Database.to_txt

`Database.to_txt(path=None, flows=True, coefficients=False, units=True, scenario='baseline', _format='txt')`

Saves the database multiple text file based on given inputs

Note:

- The function will create multiple text files carrying on the name of the matrices based on the given inputs.
 - It is suggested to keep the `units = True` so the output file can be used to parse with MARIO again.
-

Parameters

- **path** (*str*) – the path that the Excel file should be saved. If it is None, MARIO will try to use the default path and inform the user with a warning.
- **flows** (*boolean*) – if True, in the Excel file, a sheet will be created named flows containing the data of the flows
- **coefficients** (*boolean*) – if True, in the Excel file, a sheet will be created named coefficients containing the data of the coefficients
- **units** (*boolean*) – if True, in the Excel file, a sheet will be created named units containing the data of the units
- **scenario** (*str*) – defines the scenario to print out the data
- **_format** (*str*) –
 - txt to save as txt files
 - csv to save as csv files

7.7 Database parsers

7.7.1 Structured Databases

mario supports automatic parsing of following database:

- Exiobase3
- Exiobase Monetary Supply and Use
- Eurostat Supply and Use
- Eora26
- Eora single region

<code>mario.parse_exiobase_3(path[, calc_all, ...])</code>	Parsing exiobase3
<code>mario.parse_exiobase_sut(path[, calc_all, ...])</code>	Parsing exiobase mrsut
<code>mario.parse_eurostat(supply_path, use_path, ...)</code>	Parsing Eurostat databases
<code>mario.parse_eora(path, multi_region[, ...])</code>	Parsing eora databases

`mario.parse_exiobase_3`

`mario.parse_exiobase_3(path, calc_all=False, year=None, name=None, model='Database', version='3.8.2', **kwargs)`

Parsing exiobase3

Note: pxp & ixi does not make any difference for the parser.

Parameters

- **path** (*str*) – defined the zip file containing data
- **calc_all** (*boolean*) – if True, by default will calculate z,v,e after parsing

- **year** (*int*, *Optional*) – optional to the Database (just for recoding the metadata)
- **name** (*str*, *Optional*) – optional but suggested. is useful for visualization and metadata.
- **version** (*str*) – acceptable versions are:
 - 3.8.2: F_Y for the final demand satellite account
 - 3.8.1: F_hh for the final demand satellite account

Return type

mario.Database

mario.parse_exiobase_sut

mario.parse_exiobase_sut(*path*, *calc_all=False*, *name=None*, *year=None*, *model='Database'*, ***kwargs*)

Parsing exiobase mrsut

Note: mario v.0.1.0, supports only Monetary Exiobase MRSUT database.

Parameters

- **path** (*str*) – defined the zip file containing data
- **calc_all** (*boolean*) – if True, by default will calculate z,v,e after parsing
- **year** (*int*, *Optional*) – optional to the Database (just for recoding the metadata)
- **name** (*str*, *Optional*) – optional but suggested. is useful for visualization and metadata.

Return type

mario.Database

mario.parse_eurostat

mario.parse_eurostat(*supply_path*, *use_path*, *region*, *year*, *consumption_categories=['Final consumption expenditure', 'Gross Capital formation', 'Exports of goods and services']*, *factors_of_production=['Compensation of employees', 'Other taxes less other subsidies on production', 'Consumption of fixed capital', 'Operating surplus and mixed income, net', 'Taxes less subsidies on products']*, *imports=['Imports of goods and services']*, *model='Database'*, *name=None*, *calc_all=False*, ***kwargs*) → object

Parsing Eurostat databases

Note:

- this function is not generally applicable to any Eurostat table: it works only for specific table formats. Please refer to the example on the website
- first rule: it is not possible to parse file different from .xls format
- second rule: in each .xsl file, be sure data are referring to only one region
- third rule: use only “total” as stock/flow parameter, and only one unit of measure
- forth rule: supply must be provided in activity by commodity, use must be provided in commodity by activity formats
- fifth rule: only SUT table are supported

Parameters

- **supply_path** (*str*) – path to the .xls file containing the supply table
- **use_path** (*str*) – path to the .xls file containing the use table
- **region** (*str*) – name of the region: be consistent with the Eurostat names!
- **year** (*int*) – year to which the table is referring. Multiple years can be contained in the .xls files but one only can be parsed
- **consumption_categories** (*list*, *Optional*) – By default, a list of consumption categories that balance the tables according to the Eurostat criteria. The user can decide to modify them
- **factors_of_production** (*list*, *Optional*) – By default, a list of factors of production that balance the tables according to the Eurostat criteria. The user can decide to modify them
- **imports** (*list*, *Optional*) – By default, a list of imports that balance the tables according to the Eurostat criteria. The user can decide to modify them
- **name** (*str*, *Optional*) – for recording on the metadata
- **calc_all** (*bool*, *Optional*) – if True, will calculate the main missing matrices

Return type

mario.Database

mario.parse_eora

mario.parse_eora(*path*, *multi_region*, *indeces=None*, *name_convention='full_name'*, *aggregate_trade=True*, *year=None*, *name=None*, *calc_all=False*, *model='Database'*, ***kwargs*) → object

Parsing eora databases

Note:

- for *multi_region* database, only *eora26* is acceptable
 - *multi_region* database has some inconsistencies that are modified when parsed.
 - to see the modifications after parsing call 'meta_history'
-

Parameters

- **path** (*str*) – path to the zip file containing the database
- **multi_region** (*bool*) – True for *eora26* else False
- **indeces** (*str*) – in case of *multi_region* database, the *indeces.zip* file path should be given
- **name_convention** (*str*) – will take the full names of countries if *full_name* otherwise, takes the abbreviations
- **aggregate_trade** (*boolean*) – if True, will aggregated all the trades into total imports and exports in single region database
- **year** (*int*, *Optional*) – for recording on the metadata
- **name** (*str*, *Optional*) – for recording on the metadata

- **calc_all** (*boolean*) – if True, will calculate the main missing matrices

Return type

mario.Database

7.7.2 Non-Structured Databases

When databases are not structured (coming from abovementioned sources), excel or text parsers can be used. The databases in this case, should follow specific rules (examples.....)

<code>mario.parse_from_excel(path, table[, mode, ...])</code>	Parsing database from excel file
<code>mario.parse_from_txt(path, table[, mode, ...])</code>	Parsing database from text files

mario.parse_from_excel

`mario.parse_from_excel(path, table, mode='flows', data_sheet=0, unit_sheet='units', calc_all=False, year=None, name=None, source=None, model='Database', **kwargs)`

Parsing database from excel file

Note:

- This function works with a single excel that contains data & units
- Please look at the tutorials to understand the format/shape of the data

Parameters

- **path** (*str*) – defined the excel file that contains data & units.
- **table** (*str*) – acceptable options are 'IOT' & 'SUT'
- **mode** (*str*) – defined the base matrices to parse. The options are:
 - *flows*: needs [Z, Y, EY, V, E,] in a single sheet and unit in another sheet
 - *coefficients*: needs [z, Y, EY, v, e, units.txt] in a single sheet and unit in another sheet
- **data_sheet** (*str, int*) – defines the sheet index/name which the data is located (by default the first sheet is considered)
- **units_sheet** (*str, int*) – defines the sheet index/name in which the units are located (by default in a sheet named units)
- **calc_all** (*boolean*) – if True, by default will calculate z,v,e,Z,V,E after parsing
- **year** (*int, Optional*) – optional to the Database (just for recoding the metadata)
- **source** (*str, Optional*) – optional to the Database (just for recoding the metadata)
- **name** (*str, Optional*) – optional but suggested. is useful for visualization and metadata.

Return type

mario.Database

mario.parse_from_txt

`mario.parse_from_txt(path, table, mode='flows', calc_all=False, year=None, name=None, source=None, model='Database', **kwargs)`

Parsing database from text files

Note: This function works with different files to parse the io data. So every matrix & units should be placed in different txt files.

Parameters

- **path** (*str*) – defined the folder that contains data files.
- **table** (*str*) – acceptable options are 'IOT' & 'SUT'
- **mode** (*str*) – defined the base matrices to parse. The options are:
 - *flows*: needs [Z.txt, Y.txt, EY.txt, V.txt, E.txt, units.txt] in the path
 - *coefficients*: needs [z.txt, Y.txt, EY.txt, v.txt, e.txt, units.txt] in the path
- **calc_all** (*boolean*) – if True, by default will calculate z,v,e,V,E,Z after parsing
- **year** (*int*, *Optional*) – optional to the Database (just for recoding the metadata)
- **source** (*str*, *Optional*) – optional to the Database (just for recoding the metadata)
- **name** (*str*, *Optional*) – optional but suggested. is useful for visualization and metadata.

Return type

mario.Database

7.8 Calculations

7.8.1 High level matrix calculations

This function can be called inside a mario.Database object to call missing matrices for different scenarios.

<code>CoreModel.calc_all([matrices, scenario, ...])</code>	Calculates the input-output matrices for different scenarios.
<code>CoreModel.GDP([exclude, scenario, total, share])</code>	Return the value of the GDP based scenario.
<code>Database.calc_linkages([scenario, ...])</code>	Calculates the linkages in different modes

mario.CoreModel.calc_all

`CoreModel.calc_all(matrices=['z', 'v', 'e', 'Z', 'V', 'E'], scenario='baseline', force_rewrite=False, **kwargs)`

Calculates the input-output matrices for different scenarios.

Notes

By default, the function avoid the calculation of the already existing matrices in the scenario datasets. In case the user needs to overwrite the matrices, `force_rewrite = True` can be used.

It tries to find the missing data for calculating another data in a recursive process with maximum five tries.

Parameters

- **matrices** (*list*) – a list of matrices to be calculated (default values are ["z", "v", "e", "Z", "V", "E"])
- **scenario** (*str*) – the name of the scenario
- **force_rewrite** (*bool*) – False if over-write is not allowed (faster)

mario.CoreModel.GDP

`CoreModel.GDP(exclude=[], scenario='baseline', total=True, share=False)`

Return the value of the GDP based scenario.

Note: GDP based on the total V. In case that some of the items should be ignored for the calculation of the GDP (such as the imports in single region models), the user can use `exclude` argument to ignore some of the value added items.

Parameters

- **exclude** (*list, Optional*) – the items to be avoided excluded from the Value added for the calculation of the GDP.
- **scenario** (*str*) – the scenario to take
- **total** (*boolean*) – if True, it will return the total GDP. if False, it returns the sectoral GDP
- **share** (*boolean*) – if total = False, it adds a new column with sectoral share of regions

Return type

pd.DataFrame

mario.Database.calc_linkages

`Database.calc_linkages(scenario='baseline', normalized=True, cut_diag=True, multi_mode=True)`

Calculates the linkages in different modes

Note:

- Only implementable on IOTs.
 - Normalized is applicable only for single region database.
 - multi_mode is applicable only for multi region databases.
-

$$Linkages_j^{backward,direct} = \sum_{i=1}^n z_{ij}$$

$$Linkages_j^{backward,total} = \sum_{i=1}^n w_{ij}$$

$$Linkages_i^{forward,direct} = \sum_{j=1}^n b_{ij}$$

$$Linkages_i^{forward,total} = \sum_{j=1}^n g_{ij}$$

Parameters

- **scenario** (*str*) – the scenario that the linkages should be calculated for
- **normalized** (*boolean*) – normalizes linkages with average.
- **cut_diag** (*boolean*) – sets the diagonals (self consumptions) to zero.
- **multi_mode** (*True*) – work in progress

Return type

pd.DataFrame

7.8.2 Low level matrix calculations

This functions are used to calculate the matrices in mario.Database while they can be used independently outside a mario.Databases object.

<code>calc_X(Z, Y)</code>	Calculates the production vector
<code>calc_X_from_w(w, Y)</code>	Calculates Production vector from Leontief coefficients matrix
<code>calc_X_from_z(z, Y)</code>	Calculates Production vector from Intersectoral transaction coefficients matrix
<code>calc_Z(z, X)</code>	Calculates Intersectoral transaction flows matrix
<code>calc_E(e, X)</code>	Calculates satellite transaction flows matrix
<code>calc_V(v, X)</code>	Calculates Factor of production transaction flows matrix
<code>calc_M(m, Y)</code>	Calculates Economic impact matrix
<code>calc_F(f, Y)</code>	Calculates Footprint flows matrix
<code>calc_z(Z, X)</code>	Calculates Intersectoral transaction coefficients matrix
<code>calc_v(V, X)</code>	Calculates Factor of production transaction coefficients matrix
<code>calc_e(E, X)</code>	Calculates Satellite transaction coefficients matrix
<code>calc_m(v, w)</code>	Calculates Multipliers coefficients matrix
<code>calc_f(e, w)</code>	Calculates Footprint coefficients matrix
<code>calc_w(z)</code>	Calculates Leontief coefficients matrix
<code>calc_g(b)</code>	Calculates Ghosh coefficients matrix
<code>calc_b(X, Z)</code>	Calculates Intersectoral transaction direct-output coefficients matrix (for Ghosh model)
<code>calc_p(v, w)</code>	Calculating Price index coefficients vector
<code>calc_y(Y)</code>	Calculates Final demand share coefficients matrix

mario.calc_X**mario.calc_X**(*Z*, *Y*)

Calculates the production vector

$$X = z \cdot X + Y$$

Parameters

- **Z** (*pd.DataFrame*) – Intersectoral transaction flows matrix
- **Y** (*pd.DataFrame*) – Final demand flows matrix

Returns

Production flows vector

Return type*pd.DataFrame***mario.calc_X_from_w****mario.calc_X_from_w**(*w*, *Y*)

Calculates Production vector from Leontief coefficients matrix

$$x = w \cdot Y$$

Parameters

- **w** (*pd.DataFrame*) – Leontief coefficients matrix
- **Y** (*pd.DataFrame*) – Final demand flows matrix

Returns

Production flows vector

Return type*pd.DataFrame***mario.calc_X_from_z****mario.calc_X_from_z**(*z*, *Y*)

Calculates Production vector from Intersectoral transaction coefficients matrix

$$x = (I - z)^{-1}Y$$

Parameters

- **z** (*pd.DataFrame*) – Intersectoral transaction coefficients matrix
- **Y** (*pd.DataFrame*) – Final demand flows matrix

Returns

Production flows vector

Return type*pd.DataFrame*

mario.calc_Z**mario.calc_Z**(*z*, *X*)

Calculates Intersectoral transaction flows matrix

$$Z = z \cdot \hat{X}$$

Parameters

- **z** (*pd.DataFrame*) – Intersectoral transaction coefficients matrix
- **Y** (*pd.DataFrame*) – Final demand flows matrix

Returns

Intersectoral transaction flows matrix

Return type*pd.DataFrame***mario.calc_E****mario.calc_E**(*e*, *X*)

Calculates satellite transaction flows matrix

$$E = e \cdot \hat{X}$$

Parameters

- **e** (*pd.DataFrame*) – Satellite transaction coefficients matrix
- **X** (*pd.DataFrame*) – Production flows vector

Returns

Satellite transaction flows matrix

Return type*pd.DataFrame***mario.calc_V****mario.calc_V**(*v*, *X*)

Calculates Factor of production transaction flows matrix

$$V = v \cdot \hat{X}$$

Parameters

- **v** (*pd.DataFrame*) – Factor of production transaction coefficients matrix
- **X** (*pd.DataFrame*) – Production flows vector

Returns

Factor of production transaction flows matrix

Return type*pd.DataFrame*

mario.calc_M**mario.calc_M**(*m*, *Y*)

Calculates Economic impact matrix

$$M = m \cdot \hat{Y}$$

Parameters

- **m** (*pd.DataFrame*) – Multipliers coefficients matrix
- **Y** (*pd.DataFrame*) – Final Demand flows matrix

Returns

Economic impact flows matrix

Return type*pd.DataFrame***mario.calc_F****mario.calc_F**(*f*, *Y*)

Calculates Footprint flows matrix

$$F = f \cdot \hat{Y}$$

Parameters

- **f** (*pd.DataFrame*) – Footprint coefficients matrix
- **Y** (*pd.DataFrame*) – Final Demand flows matrix

Returns

Footprint flows matrix

Return type*pd.DataFrame***mario.calc_z****mario.calc_z**(*Z*, *X*)

Calculates Intersectoral transaction coefficients matrix

$$z = Z \cdot \hat{X}^{-1}$$

Parameters

- **Z** (*pd.DataFrame*) – Intersectoral transaction flows matrix
- **X** (*pd.DataFrame*) – Production flows vector

Returns

Intersectoral transaction coefficients matrix

Return type*pd.DataFrame*

mario.calc_v**mario.calc_v**(*V*, *X*)

Calculates Factor of production transaction coefficients matrix

$$v = V \cdot \hat{X}^{-1}$$

Parameters

- **V** (*pd.DataFrame*) – Factor of production transaction flows matrix
- **X** (*pd.DataFrame*) – Production flows vector

Returns

Factor of production transaction coefficients matrix

Return type*pd.DataFrame***mario.calc_e****mario.calc_e**(*E*, *X*)

Calculates Satellite transaction coefficients matrix

$$e = E \cdot \hat{X}^{-1}$$

Parameters

- **E** (*pd.DataFrame*) – Satellite transaction flows matrix
- **X** (*pd.DataFrame*) – Production flows vector

Returns

Satellite transaction coefficients matrix

Return type*pd.DataFrame***mario.calc_m****mario.calc_m**(*v*, *w*)

Calculates Multipliers coefficients matrix

$$m = v \cdot w$$

Parameters

- **v** (*pd.DataFrame*) – Factor of production transaction coefficients matrix
- **w** (*pd.DataFrame*) – Leontief coefficients matrix

Returns

Multipliers coefficients matrix

Return type*pd.DataFrame*

mario.calc_f**mario.calc_f**(*e*, *w*)

Calculates Footprint coefficients matrix

$$f = e \cdot w$$

Parameters

- **e** (*pd.DataFrame*) – Satellite transaction coefficients matrix
- **w** (*pd.DataFrame*) – Leontief coefficients matrix

Returns

Footprint coefficients matrix

Return type*pd.DataFrame***mario.calc_w****mario.calc_w**(*z*)

Calculates Leontief coefficients matrix

$$w = (I - z)^{-1}$$

Parameters**z** (*pd.DataFrame*) – Intersectoral transaction coefficients matrix**Returns**

Leontief coefficients matrix

Return type*pd.DataFrame***mario.calc_g****mario.calc_g**(*b*)

Calculates Ghosh coefficients matrix

$$g = (I - b)^{-1}$$

Parameters**b** (*pd.DataFrame*) – Intersectoral transaction direct-output coefficients matrix**Returns**

Gosh coefficients matrix

Return type*pd.DataFrame*

mario.calc_b**mario.calc_b**(*X*, *Z*)

Calculates Intersectoral transaction direct-output coefficients matrix (for Ghosh model)

$$\hat{X}^{-1} \cdot Z$$

Parameters

- **X** (*pd.DataFrame*) – Production flows vector
- **Z** (*pd.DataFrame*) – Intersectoral transaction flows matrix

Returns

Intersectoral transaction direct-output coefficients

Return type*pd.DataFrame***mario.calc_p****mario.calc_p**(*v*, *w*)

Calculating Price index coefficients vector

$$p = v \cdot w$$

Parameters

- **v** (*pd.DataFrame*) – Factor of production transaction coefficients matrix
- **w** (*pd.DataFrame*) – Leontief coefficients matrix

Returns

Price index coefficients vector

Return type*pd.DataFrame***mario.calc_y****mario.calc_y**(*Y*)

Calculates Final demand share coefficients matrix

Parameters**Y** (*pd.DataFrame*) – Final demand flows matrix**Returns**

Final demand share coefficients matrix

Return type*pd.DataFrame*

7.9 Metadata

<code>CoreModel.save_meta(path[, format])</code>	Saves the metadata in different formats
<code>CoreModel.meta_history</code>	Returns the whole history of the metadata
<code>CoreModel.add_note(notes)</code>	Adds notes to the meta history

7.9.1 mario.CoreModel.save_meta

`CoreModel.save_meta(path, format='txt')`

Saves the metadata in different formats

Parameters

- **path** (*str*) – defines the path to save metadata
- **format** (*str*) – the format of the file: ['txt', 'binary']

7.9.2 mario.CoreModel.meta_history

property `CoreModel.meta_history`

Returns the whole history of the metadata

Return type

list

7.9.3 mario.CoreModel.add_note

`CoreModel.add_note(notes)`

Adds notes to the meta history

Parameters

notes (*list*) – a list of notes to be recorded on metadata

7.10 Test

For having a simple exmaple of mario, load_test can be used.

<code>mario.load_test(table)</code>	Loads an example of mario.Database
-------------------------------------	------------------------------------

7.10.1 mario.load_test

`mario.load_test(table)`

Loads an example of `mario.Database`

table: str

type of the table. 'IOT' or 'SUT'

s

`mario.Database`

7.11 Directory

When mario needs to save an output of the model, if no path is given, files will be saved in a default directory with subfolders based on the type of the output. By default, the directory is the working directory but user can change the default directory.

`CoreModel.directory`

The default directory of the database

7.11.1 mario.CoreModel.directory

property `CoreModel.directory`

The default directory of the database

Note: by default, mario chooses the working directory as default directory

Example

Changing the default directory to a specific path called `my_directory`

```
database.directory = r'my_directory'
```

7.12 Utilities

There are some useful functions in mario that may help the user for different purposes.

`CoreModel.search(item, search)`

Searches for specific keywords in a given item

`mario.slicer(matrix, axis, **levels)`

Helps to slice the matrices

7.12.1 mario.CoreModel.search

`CoreModel.search(item, search)`

Searches for specific keywords in a given item

Parameters

- **item** (*str*) – specific level of information like Region, Satellite account, Secotr, ...
- **search** (*str*) – a keyword to search

Returns

a list of items found in the search

Return type

list

7.12.2 mario.slicer

`mario.slicer(matrix, axis, **levels)`

Helps to slice the matrices

Parameters

- **matrix** (*str*) – the matrix to be sliced
- **axis** (*int*) – 0 for rows and 1 for columns
- **levels** (*Dict['list']*) – defines the level to be sliced (according to index and columns names). for 3 level data [Region,Level,Item] and for 1 level data only [Item].

Return type

tuple, list

Example

For slicing a the final demand matrix for reg1 and sec1 on the rows and reg1 on the columns (local final demand of sec1 of reg1):

```
Y_rows = slicer(matrix= 'Y', axis= 0, Region= ['reg1'], Item= ['sec1'])
Y_cols = slicer(matrix= 'Y', axis= 1, Region= ['reg1'])

# To use the slicer
data.Y.loc[Y_rows,Y_cols]
```

7.13 Logging

In case that logging is useful for the user, the following function can be used to set the level of verbosity.

<code>mario.set_log_verbosity([verbosity, ...])</code>	Sets the formatted logging level
--------------------------------------------------------	----------------------------------

7.13.1 `mario.set_log_verbosity`

`mario.set_log_verbosity(verbosity='info', capture_warnings=True)`

Sets the formatted logging level

Parameters

- **verbosity** (*str*) – defines the level of logging such as [debug,info,warning,critical]
- **capture_warnings** (*boolean*) – if True, will capture the warnings even if the verbosity level is lower than warning

A

add_extensions() (*mario.Database method*), 22
 add_note() (*mario.CoreModel method*), 45
 add_sectors() (*mario.Database method*), 21
 aggregate() (*mario.Database method*), 20

B

backup() (*mario.Database method*), 25

C

calc_all() (*mario.CoreModel method*), 36
 calc_b() (*in module mario*), 44
 calc_E() (*in module mario*), 40
 calc_e() (*in module mario*), 42
 calc_F() (*in module mario*), 41
 calc_f() (*in module mario*), 43
 calc_g() (*in module mario*), 43
 calc_linkages() (*mario.Database method*), 37
 calc_M() (*in module mario*), 41
 calc_m() (*in module mario*), 42
 calc_p() (*in module mario*), 44
 calc_V() (*in module mario*), 40
 calc_v() (*in module mario*), 42
 calc_w() (*in module mario*), 43
 calc_X() (*in module mario*), 39
 calc_X_from_w() (*in module mario*), 39
 calc_X_from_z() (*in module mario*), 39
 calc_y() (*in module mario*), 44
 calc_Z() (*in module mario*), 40
 calc_z() (*in module mario*), 41
 clone_scenario() (*mario.Database method*), 24
 copy() (*mario.Database method*), 25

D

directory (*mario.CoreModel property*), 46

G

GDP() (*mario.CoreModel method*), 37
 get_add_sectors_excel() (*mario.Database method*), 30
 get_aggregation_excel() (*mario.Database method*), 29

get_extensions_excel() (*mario.Database method*), 30
 get_index() (*mario.CoreModel method*), 19
 get_shock_excel() (*mario.Database method*), 30

I

is_balanced() (*mario.CoreModel method*), 17
 is_hybrid (*mario.CoreModel property*), 18
 is_multi_region (*mario.CoreModel property*), 18
 is_productive() (*mario.CoreModel method*), 18

L

load_test() (*in module mario*), 46

M

meta_history (*mario.CoreModel property*), 45

P

parse_eora() (*in module mario*), 34
 parse_eurostat() (*in module mario*), 33
 parse_exiobase_3() (*in module mario*), 32
 parse_exiobase_sut() (*in module mario*), 33
 parse_from_excel() (*in module mario*), 35
 parse_from_txt() (*in module mario*), 36
 plot_bubble() (*mario.Database method*), 27
 plot_gdp() (*mario.Database method*), 26
 plot_linkages() (*mario.Database method*), 27
 plot_matrix() (*mario.Database method*), 28

R

reset_to_backup() (*mario.Database method*), 25
 reset_to_coefficients() (*mario.Database method*), 24
 reset_to_flows() (*mario.Database method*), 24

S

save_meta() (*mario.CoreModel method*), 45
 scenarios (*mario.CoreModel property*), 19
 search() (*mario.CoreModel method*), 47
 set_log_verbosity() (*in module mario*), 48
 set_palette() (*in module mario*), 26

`sets` (*mario.CoreModel property*), [19](#)
`shock_calc()` (*mario.Database method*), [25](#)
`slicer()` (*in module mario*), [47](#)
`sut_to_iot()` (*mario.Database method*), [22](#)

T

`table_type` (*mario.CoreModel property*), [19](#)
`to_excel()` (*mario.Database method*), [31](#)
`to_single_region()` (*mario.Database method*), [21](#)
`to_txt()` (*mario.Database method*), [31](#)

U

`update_scenarios()` (*mario.Database method*), [23](#)