RaMa-Scene

Release 0.0.1

Jun 29, 2018

Contents

Welcome to RaMa-Scene's docs. This documentation is split into several parts:

1. Introduction. An overview of the application inner workings.
2. API descriptors. The front-end, back-end payloads.
3. Calculations overview. Formula overview of calculations performed.
4. Deployment. Details for deploying the application.
5. Python initialise scripts. Independent scripts used for constructing files used by the application.
6. Modules. Technical details.

CHAPTER 1

Introduction

RaMa-Scene is a Django+React web-application that allows for analyzing Environmentally Extended Input-Output (EEIO) tables from EXIOBASE v3.3.

Several on-the-fly calculations are performed to generate EEIO results. The on-the-fly calculations are made possible by loading in EXIOBASE raw data into memory and employing background processing using Celery.

1.1 Implementation overview

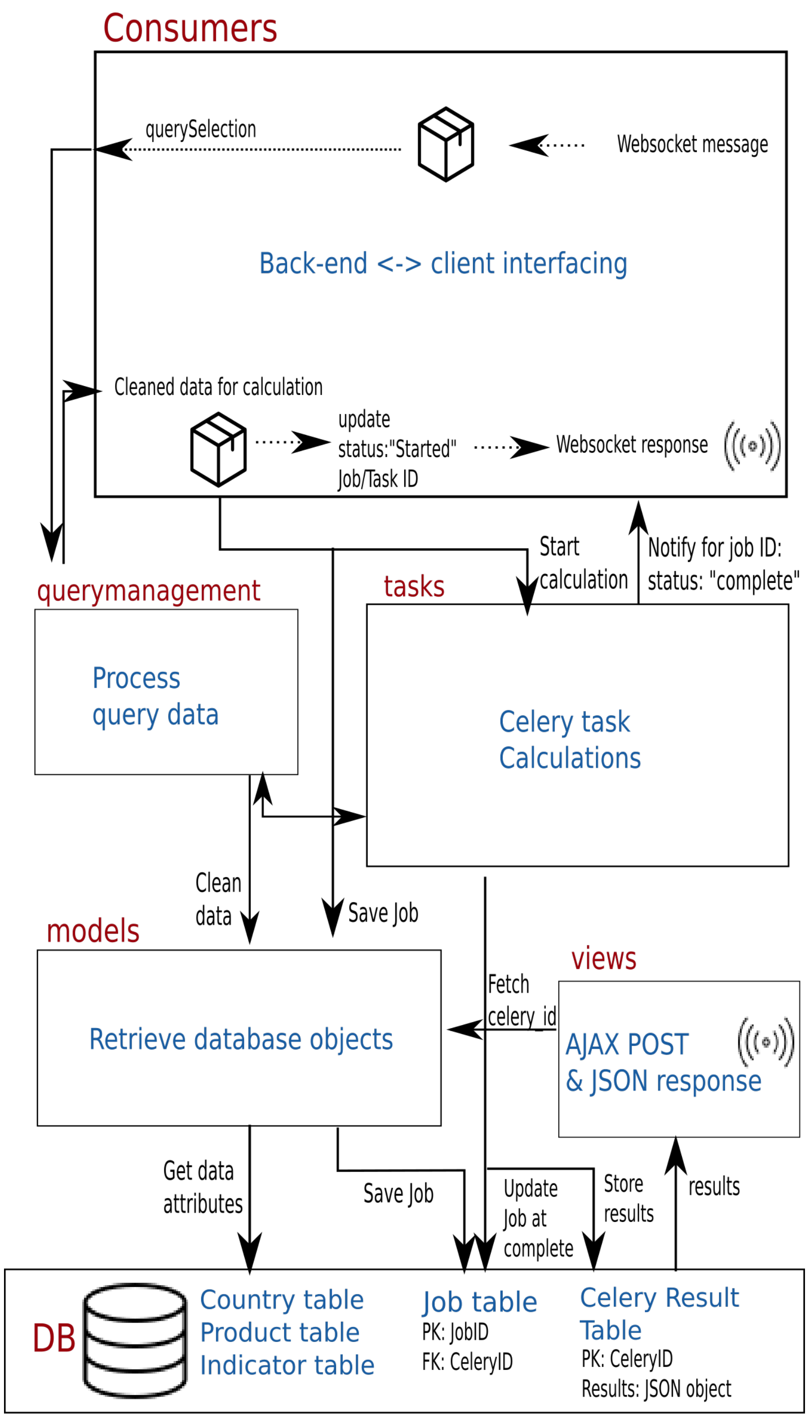
The back-end can receive two main types of requests from the front-end. A websocket-based request and an Ajax-based request.

Websockets are used for notifying the user and sending queries to the back-end, while Ajax is used to retrieve final results from the back-end.

Terminology:

- "Tasks" are used for front-end notifications of a given query and a "task" is a calculation process in Celery.
- "Jobs" are the database objects used by the back-end to remember which calculation (task) is started by which user.

  See figure below for an overview of the back-end processes.



1.2 Consumers

The module consumers is a Django Channels implementation that handles websockets. Meaning that the consumers module is one of the core communication methods between front-end and back-end. This module also invokes query management to process the queries.

1.3 Query Management

Any query received from the front-end needs to be processed in order to perform calculations and generate result data in a proper format using the query management module.

1.4 Tasks

This module is the heart of the EEIO calculations. The Tasks module implements the Celery background processing implementation that allows to process multiple calculations at the same time, but also calculations that take a long time.

1.5 Models

Models allows to insert and fetch database objects. It contains mapping coordinates for front-end and back-end communication as well as the calculation status and results per user.

1.6 Views

The views module handles the AJAX POST and JSON result response.

1.7 Analyze

For performing the actual IO calculations the analyze module is used.

CHAPTER 2

API descriptors

The React front-end has access to mapping coordinates reflecting product categories, countries and indicators. See project root static_assets:

1. final_countryTree_exiovisuals.csv
2. final_productTree_exiovisuals.csv
3. mod_indicators.csv

These mapping coordinates are not only used to render tree selectables, but also to transmit the global id's of the product categories, countries and indicators over the websocket channel. In turn the back-end handles these messages to perform calculations and store results.

API routing:

- API URL Websockets: `<domain-ip>/ramascene/`
- API URL AJAX: `<domain-ip>/ajaxhandling/`
- Interface format: JSON

Interface descriptors [websocket message to back-end]:

Stage	Instances relation	Variable name, dataType, example
Dimension	Production, Consumption	**var name: querySelection** ***JSON key: dimType, JSON value: String*** ex.: “dimType”:”Production”
Visualization	TreeMap, GeoMap	**var name: querySelection** ***JSON key: vizType, JSON value: String*** ex.: “vizType”:”TreeMap”
Filter	Product	**var name: querySelection** ***JSON key: nodesSec, JSON value: array*** ex.: “nodesSec”:”[1]”
Filter	Country	**var name: querySelection** ***JSON key: nodesReg, JSON value: array*** ex.: “nodesReg”:”[4,5]”
Filter	Indicator	**var name: querySelection** ***JSON key: ext, JSON value: array*** ex.: “ext”:”[8]”
All	→ **to back-end** [WS send]	**var name: querySelection & action** ***JSON : querySelection, JSON: action*** ex.:see table below

→ to back-end complete payload example:

```
{
        “action”:
            “start_calc”,
        “querySelection”:{
            “dimType”:”Production”,
            “vizType”:”TreeMap”,
            “nodesSec”:[1],
            “nodesReg”:[4,5],
            “ext”:[8]
            }
    }
```

Interface descriptors [websocket messages from back-end]:

Stage	Instances relation	Variable name, dataType, example
Action request status	**from Back-end** → [WS response]	**var name: action** ***JSON key: action, JSON value: string*** ex.: {“action”:”started”}
Job status	**from Back-end** → [WS response]	**var name: job_status** ***JSON key: job_status,JSON value: string*** ex.: {“job_status”:”started”}
Job status	**from Back-end** → [WS response]	**var name: job_id** ***JSON key: job_id,JSON value: int*** ex.: {“job_id”:”165”}
Job name	**from Back-end** → [WS response]	**var name: job_name** ***JSON key: job_name,JSON value: JSON*** ex.: full querySelection as names

→ **from back-end complete response example:**

```
{
        “job_id”:176,
        “action”:”check status”,
        “job_status”:”completed”,
        “job_name”:{
                        ‘nodesReg’: [‘Total’],
                        ‘vizType’: ‘TreeMap’,
                        ‘nodesSec’: [‘Fishing’, ‘Mining and quarrying’, ‘Construction’],
                        ‘dimType’: ‘Consumption’,
                        ‘extn’: [‘Value Added: Total’]
                }
}
```

Interface descriptors [AJAX response]:

Stage	Instances relation	Variable name, dataType, example
Retrieve calculation	**from Back-end** → [AJAX response]	**var name: unit** ***JSON key: name, JSON value: string*** ex.: {“Value Added”:”[M.EUR]”}
All	**from Back-end** → [AJAX response]	**var name: job_id** ***JSON key: job_id, JSON value: int*** ex.: {“job_id”:”175”}
All	**from Back-end** → [AJAX response]	**var name: rawResultData** ***JSON key: name, JSON value: array*** ex.: {“Europe”:”[1256.67]”}
All	**from Back-end** → [AJAX response]	**var name: job_name** ***JSON key: job_name, JSON value: JSON*** ex.: full querySelection as names

→ **from back-end complete response example:**

```
{
“job_id”:
        175,
“unit”:
        {“GHG emissions: Total”: “kg CO2 eq”},
“job_name”:
        {“nodesReg”: [“Europe”],
                “nodesSec”: [“Fishing”],
                “dimType”: “Production”,
                “extn”: [“GHG emissions: Total”],
                “vizType”: “GeoMap”},
“rawResultData”:
        {“Europe”: 13787995489.580374}
}
```

CHAPTER 3

Calculation overview

The following calculation explanations references the ramascene.analyze module code.

3.1 Pre-calculating the matrices

To reduce calculation times and to reduce load on the server the Leontief inverse or total requirements matrix (**L**) is precalculated according:

$$\boldsymbol{L} = (\boldsymbol{I} - \boldsymbol{A})^{-1}$$

When storing the Leontief inverse as an intermediate result, calculations on the server are reduced to simple matrix – matrix multiplication and the computational intense task of solving a system of linear equations or making a full matrix inverse can be avoided.

The matrices that are stored for further calculation are **L** , **Y** and **B**. They are stored as binary objects in the form of numpy arrays.

3.2 The four calculation routes

If environmental impacts related to final consumption are analysed it is possible to compare the environmental impacts from different points of view. We can compare the impact of consumed products between countries or between different products. This is what is called the consumption based view. Or given a certain final consumption of products we can calculate where the emissions are taking place and compare these between producing sector or countries. This is what we call the production point of view. All in all we distinguish between four different ways comparisons can be made. They are shown in Figure 1.

In each of the four calculation routes, the principal calculation that in each route is done is:

$$\boldsymbol{M} = \boldsymbol{B}\,\boldsymbol{L}\,\boldsymbol{Y}$$

The way the resulting calculations are presented are however different each time. In route 1 the calculated indicators results are presented per final consumed product. In route 2 the calculated indicator results are presented per consuming country. In route 3 the results are presented per producing country and in route 4 they are presented per produced product. For each route the details are described below.

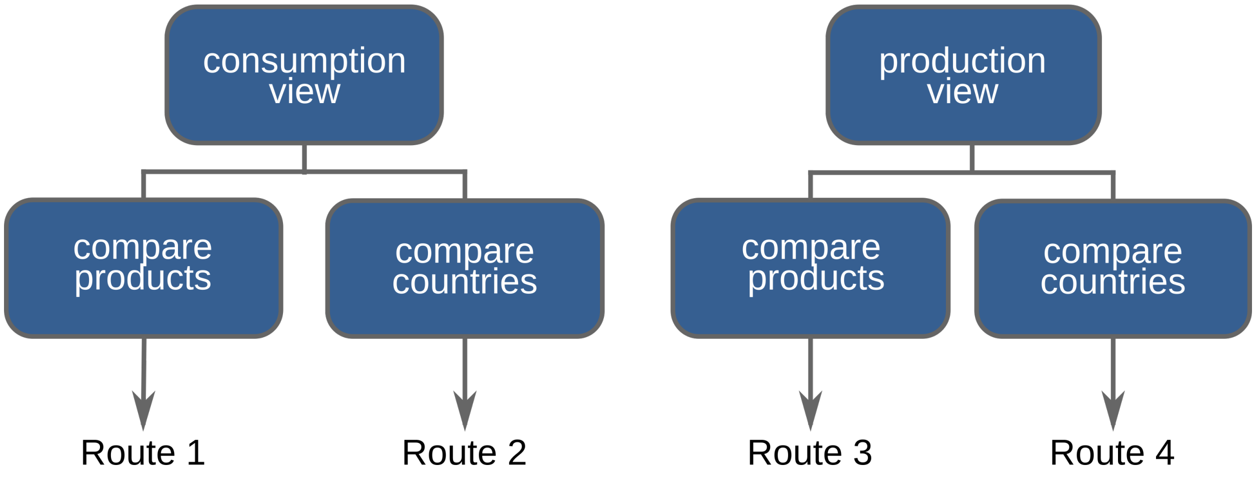


Figure 1: Consumption and production view and the corresponding calculation routes.

3.3 Route 1

In this calculation route the user can compare the environmental impacts associated with the final consumption of different products in a set of selected countries. Environmental impacts generated by production in all countries and by the production of all products are taken into account.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 1 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The symbol **Y** represents the multi-regional final demand matrix, that can be subdivided into sub-vectors that contain final demand for domestically produced products and final demand for imported products. In the final demand table there is no further subdivision into final demand by households, changes in stocks etc. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_{11} & \boldsymbol{y}_{12} & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & \boldsymbol{y}_{22} & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & \boldsymbol{y}_{32} & \boldsymbol{y}_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} \boldsymbol{y}_{11} & 0 & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & 0 & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & 0 & \boldsymbol{y}_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements. Subsequently the total final demand for each product is calculated:

$$\boldsymbol{y_s} = \boldsymbol{Y_s i}$$

Where **i** is a column vector of ones of appropriate length.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

For each of the final consumed products selected the output from each sector (**X**) needed to produce that product is calculated as:

$$\boldsymbol{X} = \boldsymbol{L}\,\widehat{\boldsymbol{y}}_s$$

At this point it is possible to make a sub-selection from **X** to select only the output in countries and sectors that are of interest to the user. For instance if we assume a three country case **X** can be expressed as:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ \boldsymbol{X}_{21} & \boldsymbol{X}_{22} & \boldsymbol{X}_{23} \\ \boldsymbol{X}_{31} & \boldsymbol{X}_{32} & \boldsymbol{X}_{33} \end{pmatrix}$$

If we're only interested in the activities taking place in country 1 as a result of the selected final consumption of products then the subselection is:

$$\boldsymbol{X}_s = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The next step is calculating the selected indicator:

$$\boldsymbol{m} = \boldsymbol{b}\,\boldsymbol{X}_s$$

where **m** is a vector of the impacts associated with each of the selected final consumed products given a certain selected region and sector where the emission takes place.

There is one last step to make. The user does not specify the geographical origin of a product. For instance if the user selects "rice" the actual product selected are "rice from Italy", "rice from Taiwan". The **m** vector with a length of 49 countries times 200 products needs to be aggregated into 200 product groups:

$$\boldsymbol{m}_{aggr} = \boldsymbol{m}\,\boldsymbol{G}$$

where **G** is an appropriate aggregation matrix.

In practice for the consumption based view the sector and region where the emission takes place is always set to all sectors and all regions. However the code allows to make sub selections.

3.4 Route 2

In route 2 the environmental impacts of final consumption is compared between countries for a selected set of products. Again environmental impacts generated by production in all countries and by the production of all products are taken into account.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 2 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The symbol **Y** represents the multi-regional final demand matrix, that can be subdivided into subvectors that contain total final demand for domestically produced products and total final demand for imported products. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_{11} & \boldsymbol{y}_{12} & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & \boldsymbol{y}_{22} & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & \boldsymbol{y}_{32} & \boldsymbol{y}_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} \boldsymbol{y}_{11} & 0 & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & 0 & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & 0 & \boldsymbol{y}_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

For each of the selected countries, the output from each sector (**X**) needed to produce that final demand for a country is calculated as:

$$\boldsymbol{X} = \boldsymbol{L}\,\boldsymbol{Y}_s$$

At this point it is possible to make a sub-selection from **X** to select only the output in countries and sectors that are of interest to the user. For instance if we assume a three country case **X** can be expressed as:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ \boldsymbol{X}_{21} & \boldsymbol{X}_{22} & \boldsymbol{X}_{23} \\ \boldsymbol{X}_{31} & \boldsymbol{X}_{32} & \boldsymbol{X}_{33} \end{pmatrix}$$

If we're only interested in the activities taking place in country 1 as a result of the selected final consumption of products then the subselection is:

$$\boldsymbol{X}_s = \begin{pmatrix} \boldsymbol{X}_{11} & \boldsymbol{X}_{12} & \boldsymbol{X}_{13} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The next step is calculating the selected indicator:

$$\boldsymbol{m} = \boldsymbol{b}\,\boldsymbol{X}_s$$

where **m** is a vector of the impacts associated with each of the countries selected final demand.

3.5 Route 3

Using this calculation route the user can compare the emissions taking place in different countries given a certain selected final demand. For instance it is possible to see in which countries emissions take place as a result of final consumption in the USA.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 3 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The **Y** symbol represents the multi-regional final demand matrix, that can be subdivided into subvectors that contain total final demand for domestically produced products and total final demand for imported products. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} y_{11} & 0 & y_{13} \\ y_{21} & 0 & y_{23} \\ y_{31} & 0 & y_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

Following the selection of final consumed products in a selected number of countries the final demand matrix is summed to get total final demand for each product:

$$\boldsymbol{y_s} = \boldsymbol{Y_s i}$$

The output needed to satisfy this final demand is subsequently calculated according:

$$\boldsymbol{x} = \boldsymbol{L y_s}$$

The vector **x** contains all possible product outputs and has a length of 49 countries times 200 products. The emissions or impact indicators are calculated by multiplying selected emission coefficients or selected indicator coefficients with the diagonalised output vector:

$$\boldsymbol{m} = \boldsymbol{b\hat{x}}$$

The vector **m** is subsequently aggregated into emission or indicators per country

$$\boldsymbol{m}_{aggr} = \boldsymbol{m}\,\boldsymbol{G}$$

where **G** is an appropriate aggregation matrix.

3.6 Route 4

Using this calculation route the user can compare the emissions associated with different product outputs given a certain selected final demand. For instance it is possible to see in which product output has the highest emissions as a result of final consumption in the USA. This calculation route starts in the same way as calculation Route 3 but the aggregation step at the end differs from route 1.

The calculation route has been designed for general application. It can calculate the environmental impacts of different products given a specific selected country selling final product or specific country where the emission takes place or specific sector where the emission takes place. However the actual implementation of route 4 takes into account that country selling product are all countries, the country where the emission takes place are all countries and the emission at all sectors are taken into account.

The calculation starts by creating the final demand vector that contains the selected products for the selected countries.

The symbol **Y** represents the multi-regional final demand matrix, that can be subdivided into subvectors that contain total final demand for domestically produced products and total final demand for imported products. Assume that there are three countries:

$$\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_{11} & \boldsymbol{y}_{12} & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & \boldsymbol{y}_{22} & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & \boldsymbol{y}_{32} & \boldsymbol{y}_{33} \end{pmatrix}$$

And the consuming country 1 and 3 have been selected then Y first becomes:

$$\boldsymbol{Y_s} = \begin{pmatrix} \boldsymbol{y}_{11} & 0 & \boldsymbol{y}_{13} \\ \boldsymbol{y}_{21} & 0 & \boldsymbol{y}_{23} \\ \boldsymbol{y}_{31} & 0 & \boldsymbol{y}_{33} \end{pmatrix}$$

where the subscript **s** stands for selected elements.

It is important to note that the user selects a product without specifying the origin of a product. For instance, if a user selects wheat as a product of interest, in the final demand vector wheat from every origin is selected i.e. wheat from Austria, wheat from Belgium etc.

Following the selection of final consumed products in a selected number of countries the final demand matrix is summed to get total final demand for each product:

$$\boldsymbol{y}_s = \boldsymbol{Y}_s\boldsymbol{i}$$

The output needed to satisfy this final demand is subsequently calculated according:

$$\boldsymbol{x} = \boldsymbol{L}\boldsymbol{y}_s$$

The vector **x** contains all possible product outputs and has a length of 49 countries times 200 products. The emissions or impact indicators are calculated by multiplying selected emission coefficients or selected indicator coefficients with the diagonalised output vector:

$$\boldsymbol{m} = \boldsymbol{b}\widehat{\boldsymbol{x}}$$

The vector **m** is subsequently aggregated into emission or indicators per country

$$\boldsymbol{m}_{aggr} = \boldsymbol{m}\,\boldsymbol{G}$$

where **G** is an appropriate aggregation matrix.

CHAPTER 4

Deployment

The web-application deployment process is based on the following documentations and is tested on Ubuntu 16.04 LTS:

1. http://masnun.rocks/2016/11/02/deploying-django-channels-using-daphne/
2. http://channels.readthedocs.io/en/stable/deploying.html
3. https://medium.com/@saurabhpresent/deploying-django-channels-using-supervisor-and-ngnix-2f9a25393eef
4. https://medium.com/@dwernychukjosh/setting-up-nginx-gunicorn-celery-redis-supervisor-and-postgres-with-django-to-run-you
5. https://www.vultr.com/docs/installing-and-configuring-supervisor-on-ubuntu-16-04

It is advised to read these guides. See the next sections for an example to get started quickly.

4.1 Install Redis [message broker]

Install redis: `sudo apt-get install redis-server`

Create a virtual environment and install the following:

```
pip3 install asgi_redis
pip3 install -U channels_redis
```

Test redis:

`redis-cli ping`

Return value should be : PONG

Make sure redis is a daemon, see redis.conf.

4.2 Install Django dependencies & prepare SQLlite

In the same virtual env., change directory towards the project root: `pip3 install -r requirements.txt`

Change directory to project folder for preparation SQLlite with Django:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Create superuser for administration purposes:

`python3 manage.py createsuperuser`

4.3 Management commands and prepare static resources

Populating database classifications:

`python3 manage.py populateHierarchies`

Adjust settings.py in project:

PATH_TO_L = '<full path to L matrix>'
PATH_TO_B = '<full path to B matrix>
PATH_TO_Y = '<full path to Y matrix>'
DEBUG = False
ALLOWED_HOSTS = [<domain>]

Install node.js (node version: 3.10.10 or higher), if not already installed:

`sudo apt-get install nodejs`

Prepare static resources:

`$npm install`

Set webpack conf settings for production:

- Configure webpack.config.js for ajax url and websocket url at webpack.DefinePlugin().
- Adjust process environment to "production" at webpack.DefinePlugin().
- Make sure that new UglifyJsPlugin() is set.

Built React bundle:

`./node_modules/.bin/webpack --config webpack.config.js`

Django collect static: `python3 manage.py collectstatic`

4.4 Install and setup nginx [HTTP and Reverse Proxy Server]

Installing nginx requires apache to be stopped if running:

```
sudo service apache2 stop
```

Install nginx:

```
sudo apt-get install nginx
```

Configure nginx, make sure proxy_pass is set to this: `http://0.0.0.0:8001`

See example configuration file `example_nginx`

Check status of nginx: `sudo nginx -t`

Allow Nginx to interact with the host machine on the network: `sudo ufw allow 'Nginx Full'`

4.5 Setting up Daphne and Celery

Make sure Daphne is installed and start daphne (in virtualenv):

```
daphne -b 0.0.0.0 -p 8001 ramasceneMasterProject.asgi:application
```

Start Celery in virtual env.:

```
celery -A ramasceneMasterProject worker -l info --concurrency=2
```

Be careful with CPU load if you raise concurrency.

Test the application to see if everything is running correct in a web-browser.

4.6 Daemonizing

Celery, Daphne need to be deamonized. For example with supervisor. See example configuration file `example_supervisord`

4.7 RaMa-Scene memory usage

The memory usage of the application is approximately 1.8G, namely due to loading raw data in memory. Loading in raw data objects in memory improves calculations speeds. The following measurements are taken:

- Daphe (Interface server for making Django available): increase from (default) 462M to 1.24G
- Celery (background processor): increase from 1.24G to 2.24G

CHAPTER 5

Python initialise scripts

Python initialise scripts provide the following features:

1. **Building mapping coordinates files for the application**
2. **Creating custom geojson and topojson files**
3. **Management commands**

Note: these scripts are not directly used by the application at runtime. If you wish to extent or develop features such as using another EEIO dataset it is recommendend to investigate how these script work. Deployment management commands are used for populating the database, see section 4.3.

5.1 Building mapping coordinates files for the application

EXIOBASE v3.3 has specific classifications that needs to map to user input. For example if Europe is selected by the user as one of the parameters for analyses, the calculation procedure uses indices corresponding to all countries belonging to Europe. In turn these calculation results need to be aggregated back into a single value for Europe. For coordinating user input to calculation procedures the RaMa-Scene application developed the following mapping CSV's :

- final_countryTree_exiovisuals.csv
- final_productTree_exiovisuals.csv

These files are read in by the script `prepare_csv.py` that in turn makes a slight modification to easily denote aggregated or disaggregated countries or product categories.

Note : see project folder python_ini/data & python_ini/devScripts.

5.2 Creating custom geojson and topojson files

For visualizations of continental data custom polygons need to be created that reflect the country mapping of EXIOBASE. A script is developed to dissolve countries that belong to a certain continent or "rest of " classification with the library GeoPandas.

Relevant scripts: `geo_dissolve_by_level.py`, `prepare_geomapping_ISO3166_3.py`, `prepare_geomapping_ISO3166_2.py`

Note : see project folder python_ini/geomapBuilds

5.3 Management commands

This application uses a sql-lite database to store mapping coordinates, user queries (Jobs) and user results (Celery results). For mapping coordinates the database needs to be populated before running the application.

The management commands uses CSV files generated by `prepare_csv.py`:

- mod_final_countryTree_exiovisuals.csv
- mod_final_productTree_exiovisuals.csv

The following command is used to populate the database `python manage.py populateHierarchies` populates the database with the mapping files needed.

Aside from database population, it is necessary to clean the database of results when needed. This can be done with the following command `python manage.py clear_models`

Note : see project folder ramascene/management/commands

CHAPTER 6

ramascene

6.1 ramascene package

6.1.1 Subpackages

ramascene.management package

Subpackages

ramascene.management.commands package

Submodules

ramascene.management.commands.clear_models module

class `ramascene.management.commands.clear_models.`**`Command`**(*stdout=None*, *stderr=None*, *no_color=False*)

Bases: `django.core.management.base.BaseCommand`

Clear database command

`handle`(**args*, ***options*)

The actual logic of the command. Subclasses must implement this method.

ramascene.management.commands.populateHierarchies module

class `ramascene.management.commands.populateHierarchies.`**`Command`**(*stdout=None*, *stderr=None*, *no_color=False*)

Bases: `django.core.management.base.BaseCommand`

Populate database with pre developed csv files residing in python_ini folder

handle(**args*, ***options*)

The actual logic of the command. Subclasses must implement this method.

ramascene.management.commands.populateHierarchies.**addCountry**(*name*, *code*, *global_id*, *parent_id*, *local_id*, *level*, *identifier*, *leaf_children_global*, *leaf_children_local*)

ramascene.management.commands.populateHierarchies.**addIndicator**(*name*, *unit*, *global_id*, *parent_id*, *local_id*, *level*)

ramascene.management.commands.populateHierarchies.**addProduct**(*name*, *code*, *global_id*, *parent_id*, *local_id*, *level*, *identifier*, *leaf_children_global*, *leaf_children_local*)

ramascene.management.commands.populateHierarchies.**getfile**(*myFile*)

ramascene.management.commands.populateHierarchies.**populate**(*data_obj*, *model_type*)

Module contents

Module contents

6.1.2 Submodules

6.1.3 ramascene.analyze module

class ramascene.analyze.**Analyze**(*product_calc_indices*, *country_calc_indices*, *indicator_calc_indices*, *querySelection*, *idx_units*, *job_name*, *job_id*, *s_country_idx*, *Y_data*, *B_data*, *L_data*)

Bases: `object`

This class contains the method for calculations

route_four()

Perform calculations according to route four.

Returns json result data

Return type json

route_one()

Perform calculations according to route one.

Returns json result data

Return type json

route_three()

Perform calculations according to route three.

Returns json result data

Return type json

route_two()
Perform calculations according to route two.

Returns json result data

Return type json

6.1.4 ramascene.consumers module

class ramascene.consumers.**RamasceneConsumer**(*scope*)
Bases: `channels.consumer.AsyncConsumer`

This class represents the Django Channels web socket interface functionality.

celery_message(*event*)
Sends Celery task status.

websocket_connect(*event*)
websocket first connection, accept immediately

websocket_disconnect(*message*)
Websocket disconnect function.

websocket_receive(*event*)
Receives message from front-end.

Tries to parse the message, if successful it will perform pre-processing steps and finally invoke Celery task.

Parameters **event** (*dict*) – message from front-end

6.1.5 ramascene.exceptions module

exception ramascene.exceptions.**ClientError**(*code*)
Bases: `Exception`

Custom exception class that is caught by the websocket receive() handler and translated into a send back to the client.

6.1.6 ramascene.models module

class ramascene.models.**Country**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Country model to store identifiers for the countries and aggregations

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

code
The country code

global_id
The global id representing the application coordinates as primary id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

identifier
an identifier determining if it is a leaf node or aggregate

leaf_children_global
the global id's of the leafs for this continent (if available)

leaf_children_local
the local id's of the leafs of this country (if available)

level
The level of hierarchy this country is in

local_id
The local id, only used if the hierarchy level is the lowest

name
The name of the country

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this country belongs to (by parent global_id)

class `ramascene.models.`**Indicator**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Indicator model to store identifiers for indicators

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

global_id
The global id representing the application coordinates as primary id

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

level
The level of hierarchy this indicator is in

local_id
The local id (unusedas there are no direct summing steps performed for the extensions)

name
The name of the indicator

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this indicator belongs to (unused as there are no direct summing steps performed for the extensions)

unit
The unit used for the indicator

class `ramascene.models.`**`Job`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Job model to store Celery jobs

exception DoesNotExist

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `django.core.exceptions.MultipleObjectsReturned`

celery_id

The unique identifier for retrieving the results of the job from Celery

completed

The date the Celery job was completed

created

The date the Celery job was created

get_next_by_created(**, field=<django.db.models.fields.DateTimeField: created>, is_next=True, **kwargs*)

get_previous_by_created(**, field=<django.db.models.fields.DateTimeField: created>, is_next=False, **kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

name

The name of the Celery job

objects = <django.db.models.manager.Manager object>

status

The status of the Celery job

class `ramascene.models.`**`Product`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Product model to store identifiers for the products and aggregations

exception DoesNotExist

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `django.core.exceptions.MultipleObjectsReturned`

code

The product category code

global_id

The global id representing the application coordinates as primary id

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

identifier

an identifier determining if it is a leaf node or aggregate

leaf_children_global

the global id's of the leafs for this product group (if available)

leaf_children_local
the local id's of the leafs of this product group (if available)

level
The level of hierarchy this product is in

local_id
The local id, only used if the hierarchy level is the lowest

name
The name of the product category

objects = <django.db.models.manager.Manager object>

parent_id
The id representing what parent this product belongs to (by parent global_id)

6.1.7 ramascene.productindexmanger module

class ramascene.productindexmanger.**ProductIndexManager**(*c_prd_ids*, *s_cntr_ids*, *p_prd_ids*, *p_cntr_ids*)

Bases: object

The ProductIndexManager provides functions to translate ids of selected countries and products into ids of the combination of selected countries and products and returns ids that cab be used directly to select columns and/or rows from final demand matrices, extension matrices and leontief inverse matrix.

The selected countries and products have to be supplied when initializing the ProductIndexManager. After initialisation no changes to the supplied ids are allowed.

Parameters

- **c_prd_ids** – ndarray 1D array containing integers indicating the ids of selected consumed products
- **s_cntr_ids** – ndarray 1D array containing integers indicating the ids of the selected countries selling final products
- **p_prd_ids** – ndarray 1D array containing integers indicating the ids of the selected produced products
- **p_cntr_ids** – 1D array containing integers indicating the ids of the selected producing countries

get_consumed_product_ids()
Get the ids of the selected consumed products

Based on the ids of the selected consumed products and the ids of the selected countries selling final products the ids of all selected products in the final demand vector are generated. It allows to make a full selection of the selected consumed products from a final demand vector. A full selection means that if the id of the product bread was selected, now the ids of bread from Italy, bread from Belgium etc are generated as long as Italy, Belgium etc are within the ids of countries selling final products. The ids are zero based.

Returns one dimensional numpy array with ids of type int

get_country_count()
Get the number of countries and regions in EXIOBASE

Returns integer object with the number of countries/regions.

get_full_selected_c_product_count()
Get the full number of consuming products selected.

Products in different countries are counted as unique items, i.e. bread from Belgium and bread from Italy consumed in a particular country are considered two items.

Returns integer object with the full count of consumed products selected.

`get_full_selected_p_product_count``()`

Get the full number of produced products selected.

Products in different countries are counted as unique items, i.e. cars produced in France and cars produced in Germany are considered two items.

Returns integer object with the full count of produced products selected.

`get_produced_product_ids``()`

Get the ids of the selected produced products

Based on the ids of the selected produced products and the ids of the selected producing countries the ids of all selected produced products are generated. It allows to make a full selection of the selected produced products from the output vector. A full selection means that if the id of the product car was selected, now the ids of car from Germany, car from France etc are generated as long as Germany, France etc are within the ids of selected producing countries. The ids are zero based.

Returns one dimensional numpy array with ids of type int

`get_product_count``()`

Get the number of products per country in EXIOBASE

Returns integer object with the number of products.

`get_selected_c_product_count``()`

Get the number of consumed products selected.

Returns integer object with the number of consumed products selected.

`get_selected_p_country_count``()`

Get the number of selected producing countries.

Returns integer object with the number of producing countries selected.

`get_selected_p_product_count``()`

Get the number of selected produced products.

Returns integer object with the number of produced products selected.

`get_selected_s_country_count``()`

Get the number of selected countries selling final products.

Returns integer object with the number of contries selling final products selected.

6.1.8 ramascene.querymanagement module

`ramascene.querymanagement.`**`clean_indicators`**`(`*idx_lst*`)`

Clean data as preprocessing step for calculation.

Clean the selected indicator by converting to integers and applying offset of -1.

Parameters **`idx_lst`** (*`list`*) – indicators

Returns indicators(processed)

Return type list

`ramascene.querymanagement.`**`clean_local_leafs`**(*a_list*)

Clean data as preprocessing step for calculation.

Clean the country or product data for calculations by splitting and converting to integers.

Parameters **`a_list`** (`str`) – country or product string of coordinates separated by #

Returns country or product list of coordinates as integers

Return type list

`ramascene.querymanagement.`**`clean_single_leafs`**(*leaf*, *OFFSET*)

Clean data as preprocessing step for calculation.

Clean the country or product data for calculations by splitting, applying offset (-1) and converting to integers.

Parameters **`leaf`** (`str`) – single country or product coordinate (non-processed)

Returns country or product list of coordinates (single element, processed)

Return type list

`ramascene.querymanagement.`**`convert_to_numpy`**(*products*, *countries*, *indicators*)

Clean data as preprocessing step for calculation.

Convert processed country,product, indicator lists to numpy array.

Parameters

- **`products`** (`list`) – pre-processed product list
- **`countries`** (`list`) – pre-processed country list
- **`indicators`** (`list`) – pre-processed indicator list

Returns numpy arrays of products, countries, indicator coordinates

Return type list

`ramascene.querymanagement.`**`get_aggregations_countries`**(*querySelection*, *result_data*)

Sum to construct aggregates results for countries.

Invoked at Celery tasks to sum values that belong to a certain aggregate.

Parameters

- **`querySelection`** (`dict`) – original querySelection from user
- **`result_data`** (`dict`) – dictionary of result_data from calculation

Returns dicitonary of result_data, but with aggregations if there are any

Return type dict

`ramascene.querymanagement.`**`get_aggregations_products`**(*querySelection*, *result_data*)

Sum to construct aggregates results for products.

Invoked at Celery tasks to sum values that belong to a certain aggregate.

Parameters

- **`querySelection`** (`dict`) – original querySelection from user
- **`result_data`** (`dict`) – dictionary of result_data from calculation

Returns dicitonary of result_data, but with aggregations if there are any

Return type dict

`ramascene.querymanagement.`**`get_calc_names_country`**`(`*country_result_data*`)`

Get name of countries.

Uses the database/model to fetch names, used inside calculation as conversion step

Parameters **`country_result_data`** (*`dict`*) – key/value pair product with key as global_id

Returns key/value pair country with key as name corresponding to querySelection global_id

Return type dict

`ramascene.querymanagement.`**`get_calc_names_product`**`(`*prod_result_data*`)`

Get name of products.

Uses the database/model to fetch names, used inside calculation as conversion step

Parameters **`prod_result_data`** (*`dict`*) – key/value pair product with key as global_id

Returns key/value pair product with key as name corresponding to querySelection global_id

Return type dict

`ramascene.querymanagement.`**`get_leafs`**`(`*product_global_ids*, *country_global_ids*`)`

Returns the leaf nodes of a given global id

Uses the database/model to fetch leaf nodes.

Parameters

- **`product_global_ids`** (*`list`*) – A list of user selected product global ids
- **`country_global_ids`** (*`list`*) – A list of user selected country global ids

Returns complete list of leaf ids (minus a offset of -1 for calculation purposes)

Return type list

`ramascene.querymanagement.`**`get_names`**`(`*prod_ids*, *country_ids*, *indicator_ids*`)`

Get name of countries, products and indicators

Uses the database/model to fetch names, used for sending selection information to front-end

Parameters

- **`prod_ids`** (*`list`*) – list of products by global id
- **`country_ids`** (*`list`*) – list of countries by global id
- **`indicator_ids`** (*`list`*) – list of indicators by global id

Returns lists of products,countries and indicators as names

Return type list

`ramascene.querymanagement.`**`identify_country`**`(`*country_id*`)`

Helper function.

Does database check on countries if the global_id the user selected is an aggregate or not

Parameters **`country_id`** (*`int`*) – global id

Returns identifier e.g. LEAF or AGG or TOTAL

Return type str

`ramascene.querymanagement.`**`identify_product`**`(`*prod_id*`)`

Helper function.

Does database check on products if the global_id the user selected is an aggregate or not

Parameters **prod_id** (*int*) – global id

Returns identifier e.g. LEAF or AGG or TOTAL

Return type str

6.1.9 ramascene.tasks module

`ramascene.tasks.`**`async_send`**(*channel_name*, *job*)

Send job message to front-end.

uses the channel_name and Job object. Send success or failure status.

Parameters

- **channel_name** (*object*) – websocket channel name
- **job** (*object*) – model object of the job

`ramascene.tasks.`**`calcOneHandler`**(*job_name*, *job_id*, *channel_name*, *ready_querySelection*, *querySelection*)

invokes Celery function.

Handler for invoking Celery method.

Parameters

- **job_name** (*str*) – the name of the job
- **job_id** (*int*) – the id of the job
- **channel_name** (*object*) – the websocket channel name
- **ready_querySelection** (*dict*) – the querySelection preprocessed (only needs convertion to numpy array)
- **querySelection** (*dict*) – the original querySelection used for aggregations at later stage

`ramascene.tasks.`**`job_update`**(*job_id*)

Update job status to completion.

Update the job status by reference job id. :param job_id: job id :type job_id: int

6.1.10 ramascene.views module

`ramascene.views.`**`ajaxHandling`**(*request*)

AJAX handler.

Checks if the request is a post. Uses from the request the task/job id to fetch the Celery unique identifier. In turn it retrieves by using the Celery unique identifier the actual results

Parameters **object** – request

Returns JSON response of result calculation

`ramascene.views.`**`home`**(*request*)

6.1.11 Module contents

`ramascene.`**`activate_foreign_keys`**(*sender*, *connection*, ***kwargs*)

Python Module Index

Index