1 Introduction 3
  1.1 Demo Output ................................................. 3
  1.2 Installation ................................................. 4
    1.2.1 Core Setup .............................................. 4
    1.2.2 Enhancement Setup ...................................... 4

2 Overview 7
  2.1 Data- and Work-Flows ........................................ 7
    2.1.1 Input Sources ........................................... 7
    2.1.2 Input from OSM or QGIS ................................. 9
    2.1.3 Input from gridder ....................................... 9
    2.1.4 Possibilities after solution ............................. 10
  2.2 Structuring .................................................. 11
    2.2.1 Repo Scope .............................................. 11
    2.2.2 Package Scope ........................................... 12
  2.3 Limitations .................................................. 13

3 Reference - Data Formats 15
  3.1 Non Spatial ................................................ 16
    3.1.1 Commodity .............................................. 16
    3.1.2 Process ................................................. 16
    3.1.3 Process-Commodity ..................................... 17
    3.1.4 Time .................................................... 17
    3.1.5 Area-Demand ........................................... 17
  3.2 Vertex ....................................................... 18
  3.3 Edge .......................................................... 18

4 Reference - Sub-Packages 21
  4.1 rivus.main ................................................ 21
    4.1.1 rivus .................................................... 21
  4.2 rivus.graph ................................................ 26
    4.2.1 to_graph ............................................... 27
    4.2.2 analysis ................................................ 28
  4.3 rivus.gridder .............................................. 29
    4.3.1 create_grid ............................................. 29
    4.3.2 extend_grid ............................................. 31
Maintainer  Johannes Dorfner, <johannes.dorfner@tum.de>

Organization  Chair of Renewable and Sustainable Energy Systems, Technical University of Munich

Version  0.1

Date  Oct 07, 2017

Copyright  This documentation is licensed under a Creative Commons Attribution 4.0 International license.
CHAPTER 1

Introduction

Extension of the welcome page.

- rivus is a MILP (mixed integer linear programming) model for multi-commodity energy infrastructure networks systems with a focus on high spatial resolution.

- It finds the minimum cost energy infrastructure networks to satisfy a given energy distribution for possibly multiple commodities (e.g. electricity, heating, cooling, . . . ).

- Time is represented by a (small) set of weighted time steps that represent peak or typical loads

- Spatial data can be provided in form of shapefiles, while technical parameters can be edited in a spreadsheet.

- Sister project to urbs which has a focus on high time resolution analysis.

1.1 Demo Output

Besides the numerical results (obtainable programatically or as an excel report), rivus is can visualize the results. You can see a unified figure of the typical result plots bellow created by rivus.main.rivus.result_figures(), which in turn is a wrapper around rivus.main.rivus.plot().

Merge was done only for better spacing in the documentation.

The unified rivus capacity output of a smaller city.

**Yellow:** Electricity network capacities

**Red:** Heat network capacities

**Brown:** Gas network capacities

**Gray:** Longitude and latitude guidelines.

**Symbols:** Diamond shapes represent energy sources. The width of the lines represents the amount of built capacity. Triangles represent energy conversion processes. Pointing upwards indicate generation, downwards indicates consumption of that commodity.
1.2 Installation

1.2.1 Core Setup

With Miniconda. (A minimal distribution of the Anaconda package manager.) If you already have Anaconda already installed, you are also good to go.

Create a clean environment with the used packages from the SciPy stack:

Get everything up and running with a one-liner (advanced, mainly for doc-build on readthedocs server):

```
conda env create -f environment.yml
```

Start with step-by-step installation process:

```
conda create --name rivus python=3.6 numpy pandas xlrd openpyxl jupyter
```

Activate your new and shiny environment as written on the console after the installation.

```
activate rivus # for windows
source activate rivus # linux
```

Add solver and geoStack. We use the with -c conda-forge the conda-forge channel, because it makes life easier. The && and -y chain the commands and suppress installation confirmation, respectively. (So that you can go grab a coffee during the installation process.):

```
conda install -y -c conda-forge pyomo glpk &&
conda install -y -c conda-forge pyshp shapely basemap &&
conda install -y -c conda-forge pyproj fiona geopy geopandas
```

1.2.2 Enhancement Setup

For optional extensions (See next section for more info.)

```
conda install -y -c conda-forge plotly
conda install -y -c conda-forge networkx
conda install -y -c conda-forge psycopg2 sqlalchemy
```

Leverage the possibilities of conda environments in your jupyter notebook. (Which is optional but really worth using it):
If you decide to use rivus.io.db, rivus.utils.notify and do not want to share your credentials with the world (highlighted), or you simply want to store some run parameters outside of version controlling. (So “check-box” changes do not disturb the actual code part.) A config file is useful (and sometimes awaited as config.json in the root of the rivus repository.) If you use it, please add /config.json to your rivus/.git/info/exclude file.

Download a template for getting you started.

```json
{
   "db": {
      "user": "postgres",
      "pass": "postgres",
      "host": "localhost",
      "base": "rivus"
   },
   "email": {
      "s_user": "your.robot@gmail.com",
      "s_pass": "thereIsASilverBullet!",
      "r_user": "your.address@gmail.com",
      "smtp_addr": "smtp.gmail.com",
      "smtp_port": 587
   },
   "solver": "glpk",
   "use_solver": true,
   "make_plot": true,
   "g_analysis": false,
   "calc_minimal": false,
   "store_db": false,
   "run_comment": "Longtime run. :codename:lyon square grid from 10-→15 edges"
}
```

In the e.g. run-script or test you can access these information with only a few lines of code:

```python
import json
config = []
with open('./config.json') as conf:
    config = json.load(conf)
...
password = config['db']['pass']
```

Should ever occur that NetworkX is too slow for a graph analysis, python-igraph can be a faster alternative. But mind the possible overhead at installation. (Esp. on Windows) As for 2017-08 following installation possibilities are available:

```bash
conda install -c conda-forge python-igraph # lin-___-osx | py2.7, py3.4<
conda install -c vtraag python-igraph # ___-win-___ | py2.7, py3.5
conda install -c marufr python-igraph # lin-win-osx | py2.7, _____
```

Go to Tutorial
CHAPTER 2

Overview

This section will:

• Get an understanding how rivus is built up and why.
• Introduce you to the sub packages and direct you to their documentation.
• Clarify what data is expected as input and what you can aspect as output.

2.1 Data- and Work-Flows

Work-flows differ from each other mainly at the beginning. From where one gets the input data. Further differences can arise if one is interested in running a multitude of optimization problems. Whatever the reason is (sensitivity analysis, structural analysis etc.) the same needs arise. The file based output will not be sufficient to manage and analyse the output effectively. And there is a good chance, that one will move to a remote server to relieve the own computer.

The following section deals with the cases of different data sources.

2.1.1 Input Sources

Most generally speaking, for working with rivus you will need the following inputs:

• Non-spatial data in a specially formatted Excel spreadsheet.
  – See Example
  – Here you can input details about costs, capacities, available commodities etc.
  – Detailed description here
• Spatial data
  – We have been using shapefiles but you can use any format which geopandas can read.
- Shapefiles are a somewhat awkward file format, but you can find an included example in mnl.

- We need an extended version of the files vertex and edge which next to the geometrical information, hold additional demand or source columns. vertex holds information about the maximum available commodity in a source vertex. edge incorporates the accumulated demand per area (building) type and per edge (street). The demand’s unit is m² as it represents the accumulated base area of different building types. This data preparation can be achieved in a run-script before model creation or with helpers manually in the data preparation process.

- Detailed description: vertex, edge

![Diagram](image)

Fig. 2.1: Import possibilities, with Open-source tools.

There are several possibilities to get the spatial input data for rivus, and conduct analysis on them.

1. Extract data from OSM (OpenStreetMap) (e.g. with the help of MapZen)
   - Extensive data preparation is needed for simplifying the geometrical structure of the vector layers. (*Currently, this workflow is under maintenance.*)
   - Example haag15 illustrates an extended version of this method.

Note: Package OSMnx could be investigated, as it proposes a very convenient gdf_from_place function. (Although it fetches one polygon, the source could serve as a good example. Alternatively, the whole shapefile phase could be skipped, with its ox.graph_from_place('Manhattan, New York, USA', network_type='drive')() function. And the edge could be calculated from that source. There are even some possibilities to simplify the resulting graph, which could be an alternative to the current skeleton based approach. Furthermore, this package already solved the automated UTM code calculation so at least, that would be of great use for us.

```python
>>> osmnx as ox
>>> city = ox.gdf_from_place('Berkeley, CA')
>>> city_xy = os.project_gdf(city)
>>> G = ox.graph_from_place('Manhattan, New York, USA', network_type='drive')
```

2. Create vector layers manually with a GIS editor. (E.g. QGIS)
   - Rather applicable for smaller research areas.
• Mind the snapping settings for vertices and edges!
• Example mnl illustrates this method.

3. Use abstract networks from the integrated rivus.gridder sub-package.
• It is integrated, so you do not need to care about data preparation. Thus ideal for getting familiar with rivus.
• If using the default, it produces a highly symmetric, homogeneous spatial input.
• See reference for usage.

2.1.2 Input from OSM or QGIS

If you choose to go with the first or second workflow, the scripts building_to_edge.py and streets_to_edge.py in the rivus.converter sub-package can help you with the data preparation. (Even if skeletron tools may not work in your environment.)

See helpful notebook, for deeper dive.

• join data from building.shp and edge.shp
• OSM street data to vertex.shp and edge.shp

The example run-script runmin.py illustrates the first two workflows.

Here is a summary, where the main part is done in 10 lines of code:

```python
import geopandas as gpd
from rivus.main.rivus import read_excel, create_model
from rivus.utils.prerun import pyomo.environ
import pyomo.opt.base import SolverFactory
...
# Non-spatial
data = read_excel(spreadsheet_path)
# Spatial
buildings = gpd.read_file(building_shp_path)
buildings_grouped = buildings.groupby(['nearest', 'type'])
total_area = buildings_grouped.sum()['total_area'].unstack()
edge = gpd.read_file(edge_shp_path).join(total_area)
vertex = gpd.read_file(vertex_shp_path)
# Model Creation and Solution
prob = create_model(data, vertex, edge)
solver = setup_solver(SolverFactory('glpk'))
solver.solve(prob, tee=True)
```

2.1.3 Input from gridder

• The notebook Square grid leads you through the input creation in depth.

Summary of the usage of gridder:

```python
from rivus.main.rivus import read_excel
from rivus.main.rivus import create_model
from rivus.gridder.create_grid import create_square_grid
from rivus.gridder.extend_grid import extend_edge_data
```
As you can see, the difference is mainly in the spatial section. The functions in gridder expose a variety of arguments to offer customization but the defaults can also be used for getting used to rivus. (E.g. above, latlon is not really needed)

2.1.4 Possibilities after solution

After the last lines of the previous code examples, you have a multitude of opportunities, what you can do with the prob ConcreteModel class. (Even a backup of your input parameters is stored in the prob.params dictionary!)

Retrieve results rivus.main.rivus showcase get_constants and get_timeseries for that. Or you can create a report in a format of an Excel spreadsheet with report.

Save the result as a serialized archive. (From which you can reload and re-run it.)

Create plots rivus.main.plot or rivus.main.result_figures expose matplotlib with all its power (and particular API...) rivus.io.fi3d and plotly will give you the tool for 3D, interactive visual data exploration. Also inside of a jupyter notebook, or exported as online, shareable website.

Conduct graph theoretical analysis rivus.graph holds the adapters so that you can leverage the opportunities offered by mainstream packages like NetworkX and python-igraph. Or export the results in mainstream graph formats like (.gml) and conduct data visualisation or analysis with a dedicated tool.

Store to PostgreSQL+PostGIS database. rivus.io.db is all about that. As it can be a rather ‘scary’ thing to jump into the world databases, a separate documentation was dedicated to help you get started and let the fears become a thing of the past.

Notify yourself Send an e-mail notification about it if this happened on a remote server.

You can find detailed description and code samples in the reference
2.2 Structuring

2.2.1 Repo Scope

Let me explain you the files/directories from which you should know the help of the following ASCII-tree:

<table>
<thead>
<tr>
<th>rivus</th>
<th>project-wise</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>OSM extract example</td>
</tr>
<tr>
<td>haag15</td>
<td>Minimalistic self-drawn example</td>
</tr>
<tr>
<td>mnl</td>
<td>Using Gridder</td>
</tr>
<tr>
<td>chessboard</td>
<td></td>
</tr>
<tr>
<td>doc</td>
<td></td>
</tr>
<tr>
<td>img</td>
<td>/html/ to check the doc locally</td>
</tr>
<tr>
<td>_build</td>
<td>run-wise for output</td>
</tr>
<tr>
<td>rivus</td>
<td>rivus package</td>
</tr>
<tr>
<td>converter</td>
<td>data preparation</td>
</tr>
<tr>
<td>graph</td>
<td>conversion + analysis</td>
</tr>
<tr>
<td>gridder</td>
<td>abstract spatial input</td>
</tr>
<tr>
<td>io</td>
<td>3D plot + DB</td>
</tr>
<tr>
<td>main</td>
<td>core model + unmoved io</td>
</tr>
<tr>
<td>tests</td>
<td>unittests</td>
</tr>
<tr>
<td>utils</td>
<td>small snippets to DRY the code</td>
</tr>
<tr>
<td><em>run</em>.py</td>
<td>project run files</td>
</tr>
<tr>
<td><em>readthedocs</em>.yml</td>
<td>config for building the docs</td>
</tr>
<tr>
<td><em>environment</em>.yml</td>
<td>config for installing dependencies on</td>
</tr>
<tr>
<td><em>readthedocs</em></td>
<td></td>
</tr>
<tr>
<td>+config.json</td>
<td>optional config file for run parameters</td>
</tr>
</tbody>
</table>

*readthedocs*.yml Configuration file for the documentation building on readthedocs server. Now it defines the python version and the conda environment configuration file for the build system. This is how we can use *napoleon_* (autodoc) to document our functions nicely.

*environment*.yml Conda environment file

**Warning:** Because of geopandas unclean installation through conda / conda-forge, it is added through pip. This is needed so that autodoc can generate the reference contents from the doc-strings.

**Note:** You could also install locally from this file with a one-liner.

```
conda env create -f environment.yml
```
Fig. 2.2: Schematic internal structure. Dashed arrow: will be re-routed in future versions. Colours are used consequently with previous figure.

### 2.2.2 Package Scope

As of version 0.2 rivus became self-contain. Its restructuring is still not complete, but the sub-packages aim to bundle the similar functions together. Reasons to do so:

- Plug-in opportunity for new functionality.
- Smaller, and thus easier maintainable files. (striven for...)
- Easier code re-usability.
- “Structural documentation”

It can be considered as a negative side-effect that import paths have grown longer.

For in-depth description see the reference of each sub-package. Here you can find a short description of each, to lead you in the good direction, if you are after something.

**Main** Core binding to the Pyomo model. You can create a whole model with just one function call. Get the resulting constants or time-series from the model.

It is also, the most *mathematically programmed* part of the code base, you will better have the mathematical reference with you if you want to dig into the mathematical model itself.

As for version 0.1, this file was the rivus script and that is why it still holds some functions, which later will be moved to the io sub-package, mixed with functions which describe mathematical rules for the Pyomo model and are not meant to be used outside of the sub-package.

**Utils** Historically, the previous python-tools functions were migrated into this sub-package. (At the time when rivus was created these were all handy functions collected/created by ojdo, now the majority is obsolete.) These will be sorted out soon, and the purpose will shift towards a container of universal code snippets.

Now deployed:

- wrapper for solver set-up
• automated parameter range generator
• e-mail notification function
• some geometrical helpers (pandashp)

But all smaller, repetitive task should find their way into this sub-package. (Create directories, get pairs of elements in a list, etc.)

**Griddler** Create and manipulate abstract versions of street and demand structures. Currently, you can create square-grids, with parametrizable features.

**IO** Prepare input for the `plotly.offline.plot()` function, with which you can generate interactive 3D plots. With a free Plotly account, you can store, embed, edit and share online your plots. But it only an option. Although, this sub-package may seem as a shiny extra, through the highly flexible API of Plotly an intuitive-interactive data exploration tool was integrated into rivus. It triggered the finding of several major bugs in the original code-base.

2D plotting will be moved here from `rivus.main`.

Besides plotting, the adapter to a PostgreSQL+PostGIS database can be found here. This module hides the most of the SQL-world and offers a convenient way to interact with the optimisation data. A separate documentation was dedicated to help users started, and document the now awaited data structure.

**Graph** Convert the resulting tabular data, which represent the built commodity carrier grids (electricity grid, Gas pipelines, district heating/cooling grid etc.) into a graph (network) format of either NetworkX or python-igraph. Moreover, the file export functions of these libraries were bridged into a module in this sub-package. (Preferred file format is `.gml` which is supported by all common graph analysis tools. E.g. Gephi the Open-source de-facto tool for advanced graph visualisation and analysis.)

After the data is in their expected format, both NetworkX and python-igraph offer very advanced opportunities to analyse graphs. You can look up what you need in their documentation. Nevertheless, some basic analyse wrapper is provided to get the result for the most common questions about graph connectivity.

**Converter** The only sub-package, which was not intended to be used by import, but as a container for separate scripts which facilitate data preparation from real-world street network data sources.

**Tests** Yes, the unit tests are located here. (What a surprise…) One of the youngest members of the rivus sub-packages. It should not be necessary to emphasize the importance of testing, but still again and again the joy of implementation takes away the focus from actually test the written code.

As most of the bugs encountered during my work with rivus could have been avoided with simple unit tests, I tried to set an example and write tests in parallel to each new function. Anyway, there is a long way to go, but it is worth the energy.

### 2.3 Limitations

Two citations to keep in mind, when working with mathematical programming:

The purpose of mathematical programming is insight, not numbers.

—Arthur M. Geoffrion (in 1976)

Essentially, all models are wrong, but some are useful.
**Warning:** As for the current state, *rivus* does not consider already existing energy infrastructure networks. Thus the solution always assumes a **from scratch** planning. (Feature is planned to be integrated into the logic later.)

**Warning:** As for now, *rivus* does **not handle storage** in any way.

**Warning:** *rivus* estimates the cost-capacity functions of commodity transfer building blocks to be linear and continuous. (E.g. instead of handling discrete pipe diameters and prices, a linear function is used instead.)

**Note:** Research conducted with *rivus* was on the **urban level**. Theoretically, there is no barrier for the model to reach for bigger structures, but pragmatically a performance boost would be needed to push the project in the direction of applicability to bigger or more detailed problems.

**Note:** There is no n-1 redundancy built in the model yet.
As rivus expects its input variables from different sources, this section shall help you to get along with the parameters.

Fig. 3.1: Input sources and their data-flow until the main variables. (Vertex, Edge, Data)
3.1 Non Spatial

You can retrieve non-spatial data from a spreadsheet or from a database. As the spreadsheet was is the standard data input format, we will discuss that in the following section. You can find examples (templates) in the /data/haag15/ or /data/mnl/ data folders.

This summary of the columns shall ease your understanding of those input variables. (Extracted from the tool-tip info in the haag15 project.)

Note:

- If a cell does not have meaningful value in the record, that should be marked with the =$N/A$ formula. Not empty cell.
- Names of the sheets and columns must be preserved, as they are hard-coded into the parser.
- Do not forget to check the unit of your inputs!
- Do not leave ‘zombie’ record (e.g. a process without listed commodity).

3.1.1 Commodity

cost-inv-fix Fixed investment costs [€/m] Capacity-independent investment costs for pipe/cable to transmit that commodity.
cost-inv-var Variable invest costs [€/kW/m] Capacity-dependent investment costs for transmission capacity of a commodity from one vertex to another.
cost-fix Variable fixed costs [€/kW/m] Capacity-dependent fixed costs for maintaining transmission capacity.
cost-var Purchase costs [€/kWh] Cost for buying that commodity at source vertices, if any exist in the vertex_shapefile.
loss-fix Fixed loss fraction [kW/m] Powerflow-independent loss of energy per meter of transmission length through the network. The fixed loss is calculated by (length * loss-fix).
loss-var Variable power loss [1/kW/m] Relative loss term, dependent on input power flow through a “pipe”: Ingoing power flow per edge is multiplied by (1 - length * loss-var)
cap-max Maximum capacity [kW] Maximum possible transmission capacity per edge.
allowed-max Maximum allowed generation Limits the net amount of generation of this commodity (e.g. CO2). Note that processes that consume a commodity (e.g. CCS) can reduce the net amount.

3.1.2 Process

cost-inv-fix Fixed investment costs [€]

Up-front investment for building a plant, independent of size. Has value zero mainly for small-scale technologies.
cost-inv-var Specific investment costs [€/kW]

Size-dependent part for building a plant.
**cost-fix**  Specific fixed costs [€/kW]

Size-dependent part for maintaining a plant.

**cost-var**  Variable costs [€/kWh]

Operational costs to produce one unit of output, excluding fuel costs. Has value zero e.g. for PV or wind turbines (or if no sources are available).

**cap-min**  Minimum capacity [kW]

Smallest size a plant is typically available in. Has value zero for domestic technologies.

**cap-max**  Maximum capacity [kW]

Biggest capacity a plant typically is available in.

### 3.1.3 Process-Commodity

**ratio**

Input/output ratio

Flows in and out of processes, relative to 1 unit of throughput. For CO2, unit is kg/kWh (for example)

### 3.1.4 Time

**weight**

Time step weight [hours]

Length of time step in hours. Sum of all weights == 8760

**Elec**

Scaling factor Elec [1]

Relative scaling factor of demand ‘Elec’ per time step. Interpret like y-values of a normalised annual load duration curve.

**Heat**

Scaling factor Heat [1]

Relative scaling factor of demand ‘Heat’ per time step. Interpret like y-values of a normalised annual load duration curve.

### 3.1.5 Area-Demand

**peak**

Building peak demand [kW/m²]

Peak demand of building type (must be present in building_shapefile) normalised to building area. Annual demand is encoded in time step weights on sheet Time.
3.2 Vertex

The examples are given with the help of the Gridder sub-package, but the that depicts very well what you should see in a shapefile’s attribute list. (Excluding the special geometry column of course.)

You can also check it out quickly:

```python
vdf, edf = create_square_grid()
extend_edge_data(edf)
vert_init_commodities(vdf, ['Elec', 'Heat', 'Gas'], [('Elec', 0, 5000),])
print(vdf.head())
```

Should give you:

<table>
<thead>
<tr>
<th>geometry</th>
<th>Vertex</th>
<th>Elec</th>
<th>Heat</th>
<th>Gas</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINT (11.66842 48.26739)</td>
<td>0</td>
<td>5000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>POINT (11.66976700131334 48.26738999211108)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>POINT (11.66842 48.26828931656865)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>POINT (11.66976702494603 48.26828930867949)</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

3.3 Edge

The examples are given with the help of the Gridder sub-package, but the that depicts very well what you should see in a shapefile’s attribute list. (Excluding the special geometry column of course.)

You can also check it out quickly:

```python
vdf, edf = create_square_grid()
extend_edge_data(edf)
vert_init_commodities(vdf, ['Elec', 'Heat', 'Gas'])
print(edf.head())
```

Should give you:

<table>
<thead>
<tr>
<th>geometry</th>
<th>Edge</th>
<th>Vertex1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINESTRING (11.66842 48.26739, 11.66976700131334 48.26738999211108)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LINESTRING (11.66842 48.26828931656865, 11.669...</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LINESTRING (11.66842 48.26739, 11.66842 48.268...</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>LINESTRING (11.66976700131334 48.2673899921110...</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
You can get an overview of the purpose of the sub-packages from the previous section. The (detailed) description of all user-relevant functions in rivus are listed below. In addition to the documentation of the individual functions, a brief description is provided per sub-package or module to enlighten the used packages or design logic.

**Note:**

**Prerequisites:** In the code examples, we assume that you have a installed rivus (git clone) and the required packages for your work. (conda install see in installation)

### 4.1 rivus.main

#### 4.1.1 rivus

The formulation of the MILP model is implemented with the help of Pyomo’s ConcreteModel. That means, the needed constrains are formulated with the help of python functions. These have descriptive names following the *_rule or *_balance naming convention. In general, you do not have to touch them.

As for now, the object returned by create_model() (often named as prob) serves as the base type of input for the other function. This may change in future versions. But as for now, it is important you get familiar with its format. Following attributes are directly accessible and esp. interesting in the first round.

- peak : (Pandas.DataFrame) - Maximum demands per edge.
- r_in, r_out : (Pandas.Series) - Process-Commodity ratios split by direction.
- params : (dict)
  - commodity : processed Excel sheet
rivus Documentation, Release 0.1

- process - processed Excel sheet
- hub - filtered version of process
- process_commodity - processed Excel sheet
- time - processed Excel sheet
- area_demand - processed Excel sheet
- vertex - processed spatial input
- edge - processed spatial input

Warning: Following functions will be migrated to the io sub-package:
- read_excel
- plot - result_figures
- save - load
- save_log

Following function will be migrated to rivus.main.getters:
- get_entity - get_entities - list_entities
- get_onset_names
- get_constants
- get_timeseries

Modelling

rivus: optimization model for distributed urban energy systems

rivus optimizes topology and size of urban energy networks, energy conversion.

create_model (data, vertex, edge, peak_multiplier=None, hub_only_in_edge=True)
Return a rivus model instance from input file and spatial input.

Parameters

- spreadsheet – Excel spreadsheet with entity sheets Commodity, Process, Process-Commodity, Time and Area-Demand
- vertex – DataFrame with vertex IDs as column ‘Vertex’ and other columns named like source commodities (e.g. ‘Gas’, ‘Elec’, ‘Pellets’), containing source vertex capacities (in kW)
- edge – DataFrame woth vertex IDs in columns ‘Vertex1’ and ‘Vertex2’ and other columns named like area types (in spreadsheet/Area-Demand), containing total areas (square metres) to be supplied

Returns Pyomo ConcreteModel object

line_length (line)
Calculate length of a line in meters, given in geographic coordinates.
**Parameters** line – a shapely LineString object with WGS 84 coordinates

**Returns** Length of line in meters

### IO

Following functions will be migrated to the `io` sub-package: rivus: optimization model for distributed urban energy systems

**read_excel** *(filename)*

Read Excel input file and prepare rivus input data dict.

Reads an Excel spreadsheet that adheres to the structure shown in the example dataset `data/mnl/mnl.xlsx`. Must contain

**Parameters** filename – filename to an Excel spreadsheet.

**Returns** a dict of 6 DataFrames, one for each sheet

**plot** *(prob, commodity, plot_demand=False, mapscale=False, tick_labels=True, annotations=True, buildings=None, shapefiles=None)*

Plot a map of supply, conversion, transport and consumption.

For given commodity, plot a map of all locations where the commodity is introduced (Rho), transported (Pin/Pot/Pmax), converted (Epsilon_*) and consumed (Sigma, peak).

**Parameters**

- prob
- commodity
- plot_demand – If True, plot demand, else plot capacities
- mapscale – If True, add mapscale to plot (default: False)
- tick_labels – If True, add lon/lat tick labels (default: True)
- annotations – If True, add numeric labels to graph (default: True)
- buildings – tuple of (filename to shapefile, boolean) if true, color buildings according to attribute column “type” and colors in constan rivus.COLORS; else use default COLOR[‘building’] for all
- shapefiles – list of dicts of shapefiles that shall be drawn by basemap function readshapefile. is passed as **kwargs

**Returns** fig – the map figure object

**result_figures** *(prob, file_basename, buildings=None, shapefiles=None)*

Call rivus.plot with hard-coded combinations of plot_type and commodity.

This is a convenience wrapper to shorten scripts. TODO: Generalise so that no hard-coding of commodity names is needed.

**Parameters**

- prob – a rivus model instance
- file_basename – filename prefix for figures
• **buildings** – optional filename to buildings shapefile

**Returns** Nothing

**report**(prob, filename)

Write result summary to a spreadsheet file.

Create a concise result spreadsheet with values of all key variables, including costs, pipe capacities, process and hub capacities, source flows, and process input/output/throughput per time step.

**Parameters**

• **prob** – a rivus model instance

• **filename** – Excel spreadsheet filename, will be overwritten if exists

**Returns** Nothing

**save_log**(result, filename)

Save urbs result and solver information to a log file.

**Parameters**

• **result** – as returned by the solve method of a solver object

• **filename** – log file to be written

**Returns** Nothing

**save**(prob, filename)

Save rivus model instance to a gzip’ed pickle file

Pickle is the standard Python way of serializing and de-serializing Python objects. By using it, saving any object, in case of this function a Pyomo ConcreteModel, becomes a twoliner. GZip is a standard Python compression library that is used to transparently compress the pickle file further. It is used over the possibly more compact bzip2 compression due to the lower runtime. Source:

**Parameters**

• **prob** – a rivus model instance

• **filename** – pickle file to be written

**Returns** Nothing

**load**(filename)

Load a rivus model instance from a gzip’ed pickle file

**Parameters** **filename** – pickle file

**Returns** **prob** – the unpickled rivus model instance

**Getters**

Following function will be migrated to rivus.main.getters

rivus: optimization model for distributed urban energy systems

rivus optimizes topology and size of urban energy networks, energy conversion.
get_entity \((\text{instance}, \text{name})\)
Return a DataFrame for an entity in model instance.

Parameters

- \text{instance} – a Pyomo ConcreteModel instance
- \text{name} – name of a Set, Param, Var, Constraint or Objective

Returns a single-columned Pandas DataFrame with domain as index

get_entities \((\text{instance}, \text{names})\)
Return one DataFrame with entities in columns and a common index.

Works only on entities that share a common domain (set or set_tuple), which is used as index of the returned DataFrame.

Parameters

- \text{instance} – a Pyomo ConcreteModel instance
- \text{names} – list of entity names (as returned by list_entities)

Returns a Pandas DataFrame with entities as columns and domains as index

list_entities \((\text{instance}, \text{entity_type})\)
Return list of sets, params, variables, constraints or objectives.

Parameters

- \text{instance} – a Pyomo ConcreteModel object
- \text{entity_type} – “set”, “par”, “var”, “con” or “obj”

Returns DataFrame of entities

Example

```python
>>> data = read_excel('data-example.xlsx')
>>> model = create_model(data, range(1,25))
>>> list_entities(model, 'obj')
Name       Description Domain
---        ----------- ------
obj minimize(cost = sum of all cost types)   []
[1 rows x 2 columns]
```

get_onset_names \((\text{entity})\)

Example

```python
>>> data = read_excel('data-example.xlsx')
>>> model = create_model(data, range(1,25))
>>> get_onset_names(model.e_co_stock)
['t', 'sit', 'com', 'com_type']
```

get_constants \((\text{prob})\)
Retrieve time-independent variables/quantities.
Usage: costs, Pmax, Kappa_hub, Kappa_process = get_constants(prob)

Parameters prob – a rivus model instance

Returns (costs, Pmax, Kappa_hub) tuple

get_timeseries(prob)
Retrieve time-dependent variables/quantities.

Usage: source, flows, hubs, proc_io, proc_tau = get_timeseries(prob)

Parameters prob – a rivus model instance

Returns (source, flows, hubs, proc_io, proc_tau) tuple

4.2 rivus.graph

Comparison of the common graph analysis tools:

• NetworkX (preferred):
  – +/− Pure python implementation.
  – + Widely used and tested.
  – + Docs are quite good.
  – + Easy (platform independent) installation
  – - Slower than igraph (and graph-tools)

• python-igraph (fall-back):
  – + C based with python wrappers.
  – + Mature library package.
  – + Included for speed and so for scalability.
  – × Docs are OK.
  – - Windows install can be somewhat tedious (with unofficial wheel files). But it works.

• graph-tools: (eventually added in the future, if there is reeealy big need for efficiency..)
  – + Self proclaimed: fastest in graph analyses
  – - Not really windows user friendly (docker install should be tested)

These all facilitate pretty advanced graph theoretical analysis. Moreover, the file export functions of the used libraries were bridged through the to_graph module. Preferred file format is .gml which is supported by all common graph analysis tools. As such, specialized, stand-alone tools for graph visualisation and analysis can be used. E.g. Gephi with its exceptional geolayout plug-in can be of tremendous help for deeper graph visualization and analysis.
4.2.1 to_graph

Functions to convert tabular data to popular python graph structures

to_igraph (vdf, edf, pmax, comms=None, peak=None, save_dir=None, ext='gml')

Convert Data from (Geo)DataFrames to python-igraph’s Graph class Each commodity gets its own graph. Weights are derived from built capacity.

Parameters

• **vdf** ([Geo]DataFrame) – Holding Vertex Data id=Vertex and Commodity Sources as columns

• **edf** ([Geo]DataFrame) – Holding (V1,V2) Multi-indexed Edge data To be sure, that all edges are created.

• **pmax** (DataFrame) – Commodities as columns with max capacity per edge returned by rivus.get_constants()

• **comms** (iterable, optional) – Names of the commodities from which we build the graphs. (Each as separate graph.) If omitted, the columns of pmax will be used.

• **peak** (DataFrame, optional) – Commodities as columns with demands in t_peak time-step. Calculated in main.rivus

• **save_dir** (path string, optional) – Path to a dir to save graphs as ext Path preferably constructed using the os.path module. If dir does not exit yet, it will be created.

• **ext** (str, optional) – Description


Returns **list** – List of igraph.Graph objects in order of **comms**. Graphs are undirected and weighted.

Example

::

    _, pmax, _, _ = get_constants(prob) graphs = to_igraph(vertex, edge, pmax, ['Gas', 'Heat'])

to_nx (vdf, edf, pmax, comms=None, save_dir=None)

Convert to networkx graph representation

Parameters

• **vdf** ([Geo]DataFrame) – Holding Vertex Data id=Vertex and Commodity Sources as columns

• **edf** ([Geo]DataFrame) – Holding (V1,V2) Multi-indexed Edge data To be sure, that all edges are created.

• **pmax** (DataFrame) – Commodities as columns with max capacity per edge returned by rivus.get_constants()
• **comms**(iterable, optional) – Names of the commodities from which we build the graphs. (Each as separate graph.) If omitted, the columns of `pmax` will be used.

• **save_dir**(path string, optional) – Path to a dir to save graphs as GML. Path preferably constructed using the `os.path` module. If dir does not exist yet, it will be created.

**Returns** list – nx_graph objects in accordance with input `comms` or all commodities found in `pmax.columns`

**Example**

```python
_: _, pmax, _, _ = get_constants(prob) graphs = to_nx(vertex, edge, pmax, ['Gas', 'Heat'])
```

**Note:** `nx.from_pandas_dataframe()` was also investigated for conversion, but it is a bit slower and does not improve code quality in my opinion.

### 4.2.2 analysis

Functions to hold dedicated analysis runs on the graph objects. `networkx` is the de-facto graph package, but `igraph` compatibility is also eligible.

**minimal_graph_anal**(graphs, calc_spanning=True, graph_package='NX')

Showcase interoperable connectivity analysis.

**Parameters**

- **graphs**(list) – networkx or igraph Graph objects. Awaited is the resulting ITERABLE of `rivus.graph.to_graph` functions.

- **calc_spanning**(bool, optional) – Default: True, sets whether to investigate: Is the graph of the built commodity also a minimal spanning tree of the street network?

- **graph_package**(str, optional) – Default: ‘NX’, Accepted: ‘NX’ or ‘IGRAPH’. To ease the decision of which package is used.

**Returns**

list of dicts per graph –

- commodity
- is_connected
- connected_components (number of them)
- is_minimal (if `calc_spanning` is True)
4.3 rivus.gridder

The birth of this module originates to the intention to regard the linear optimization model as a natural phenomenon. As usual approach in natural sciences, we abstract the specific question in focus, removing the “noise” of real-world data and homogenize the otherwise heterogeneous input data.

After gaining understanding of the highly symmetric model, the noise and asymmetries can be added back in an iterative process. Thus testing the validity of our findings.

As the first approach, a symmetric grid generator is implemented, which can be highly parametrized and can create input for `create_model()` only in few lines of code.

![Fig. 4.1: Adding noise to the grid. (x, and y edge length can be also set separately.)](image)

Moreover, the possibility to have an input generator integrated into `rivus` itself has proven itself useful in many ways. Testing the implemented paradigms can be more intuitive in a symmetric environment, and the ability to rapidly showcase little optimization projects can serve as a demonstration tool accompanying relevant lectures.

**Note:** A highly symmetric square grid may seem far-fetched from real-life street structures. However, remember the grid street plan which has been around since the Greeks and is to be found all over the world. So, it is maybe not such a bad starting point as considered it may seem at the first glance.

Short example set of cities [Google Maps Set](#).

### 4.3.1 create_grid

`create_square_grid` (origo_lation=(48.26739, 11.66842), num_edge_x=1, num_edge_y=1, dx=100, dy=100, noise_prop=0.0, epsg=None, match=0)

Create chessboard grid with edges and vertices on WGS84 surface with vincenty distance calculation lat ~ x, lon ~ y

**Parameters**
• **origo_latlon** *(tuple, optional)* – WGS84 latlon coordinates of the bottom left grid point defaults to some the TUM-ENS dep.

• **num_edge_x** *(int, optional)* – how many edges horizontally

• **num_edge_y** *(None, optional)* – How many edgery vertically

• **dx** *(int, optional)* – length of the horizontal edges (in meters)

• **dy** *(None, optional)* – length of the vertical edges (in meters)

• **noise_prop** *(float, optional)* – 0.0 to MAX_NOISE < 1.0 effectively a relative missplacement radius

• **epsg** *(int, optional)* – If a valid epsg code which is supported py pyproj, the coordinates are calculated in the carthesian UTM CRS and then transformed into epsg4326 (latlon). If None or omitted, then the coordinates are calculated directly in epsg4326 with vincenty’s formula for distance and the grid lines up with the North and East directions

• **match** *(enumerated values, optional)* –

  0 [vertices and edges are matched by the logic of generation] (faster as less calculation is needed.)

  1 [matching is done geographically] with pandashp helper (slower, but flexible)

• **Return** *(vertices, edges) As GeoDataFrames* – vertices : [geometry, Vertex] edges : [geometry, Edge, Vertex1, Vertex2]

• **Indexing** –

  (6)04(7)05(8)  7 9 11

  (3)02(4)03(4)  6 8 10

  (0)00(1)01(2)

**get_source_candidates** *(vdf, dim_x, dim_y, logic='sym')*

Calculate the set of indexes of the vertices, which are worth testing as source vertex in a single commodity case. A square grid is assumed. “Worth” means: The minimal set of vertices which cover the main symmetrical positions.

**Parameters**

• **vdf** *(pandas DataFrame)* – The vertex frame. (Created by create_square_grid())

• **dim_x** *(int)* – Number of vertices along the x axis.

• **dim_y** *(int)* – Number of vertices along the y axis.

• **logic** *(str, optional default='sym')* – what kind or source candidates are looked for. sym : Minimal(ish) set of vertices based on symmetry.

  E.g. here the indices marked with * are selected. 18, 19, 20, 21, 22, 23 12, 13, 14, 15, 16, 17 *6, *7, *8, 9, 10, 11 *0, *1, *2, 3, 4, 5

  **extrema**: Pairs of vertices possibly further away from each other. Say: combination of the corners.
**Returns** List – smy : 1D list [1,2,6,7,8] extrema, center : 2D list - list of lists [[0, 23], [0, 5], [0, 18]]

### 4.3.2 extend_grid

**extend_edge_data** *(edge, sorts=None, inits=None, strat='equal', strat_param=None)*

Add demand data to the edges in a (Geo)DataFrame

**Parameters**

- **edge_df** *(Geo)DataFrame* – edge dataframe to be extended
- **sorts** *(list of str, optional)* – The names of new columns (extensions) Defaults to [‘residential’]
- **inits** *(list of int/float, optional)* – The parameter values, matching to sorts argument. Defaults to [1000] for each sort.
- **strat** *(str, optional [TODO now only 'equal' has an effect]) – How the data values will be created + ‘equal’ - all edge demand is the same + ‘linear’ - linearly decreasing + ‘exp’ - exponentially decreasing + ‘manual’ - provide mapper in strat_param
- **strat_param** *(optional [TODO no implemented yet]) – Parameter for linear | exp | manual strategies. + ‘equal’ - None - no effect + ‘linear’ - minimum (lowest demand) + ‘exp’ - minimum (lowest demand) + ‘manual’ - function/dict to fetch value per edge

**Raises**

- ValueError
- If inputs differ from awaited
- Usage

---

**sorts** = (‘residential’, ‘other’)
- **inits** = (1000, 800)
- **extend_edge_data**(edge, sorts=sorts, inits=inits)

**vert_init_commodities** *(vertex_df, commodities, sources=None, inplace=True)*

Add commodity columns to the vertex DataFrame with zeros to vertices without commodity source and source capacity at the vertices provided by *sources*.

**Parameters**

- **vertex_df** *(Geo)DataFrame* – vertex dataframe input
- **commodities** *(list of str)* – Like (‘Elec’, ‘Gas’, ‘Heat’)
- **sources** *(list of tuples, optional)* – Init the source nodes. Tuple form:(Commodity, Index, Value)
- **inplace** *(Boolean, default: True)* – If False, vertex_df is not changed and the result is returned.

**Returns** None or DataFrame – DataFrame if inplace is False
4.4 rivus.io

4.4.1 rivus.db

To advocate the possibilities provided by a good database connection, a throughout description of the set-up process is documented in rivus-db. There you can find help from the entry level (install, create database) to more advanced topics (queries, data archive).

In this module presents a convenient way to interact with your PostgreSQL database.

```python
from sqlalchemy import create_engine
from rivus.io import db as rdb
engine = create_engine('postgresql://postgres:pass@localhost/rivus')

this_run = dict(comment='testing graph table and features with networx',
                 profiler=profile_log)

rdb.store(engine, rivus_model, run_data=this_run)
```

Example queries with results and short descriptions are part of the separate documentation.
df_from_table (engine, fname, run_id)

Extract data form the database into a dataframe in a form, that is common during the rivus workflow.

Implemented DataFrames:

- rivus_model.params[] dataframes:
  - process
  - commodity
  - process_commodity
  - edge
  - vertex
  - time
  - area_demand
- get_timeseries dataframes:
  - source
- get_constants dataframes:
  - cost
  - pmax
  - kappa_hub
  - kappa_process

Parameters

- **engine** (sqlalchemy engine whit psycopg2 driver) – For managing connection to the DB.
- **fname** (str) – One of the implemented dataframes. (See summary.)
- **run_id** (int) – run_id of an initialized run row in the DB. You could query the run table for e.g. start date, or join it vertex table and execute a geographical query and get the run_id(s) you want to work with

Returns DataFrame or Series – depending on the data’s dimensions. Only cost returns a Series to be consequent with get_constants.

init_run (engine, runner='Havasi', start_ts=None, status='prepared', outcome='not_run', comment=None, plot_dict=None, profiler=None)

Initialize the run table with basic info.

Parameters

- **engine** (sqlalchemy engine whit psycopg2 driver) – For managing connection to the DB.
- **runner** (str, optional) – Person’s name/identifier who created(executed) the data(process).
rivus Documentation, Release 0.1

• **start_ts** (*datetime.datetime, optional*) – Timezone-less datetime object. If omitted, .now() will be used.

• **status** (*str, optional*) – One of the following strings: `'prepared'` (default) | `'run' | `'error`

• **outcome** (*str, optional*) – One of the following strings: `'not_run' (default) | 'optimum' | 'optimum_not_found' | 'error'

• **comment** (*str, optional*) – Any text based comment. (No length limit.)

• **plot_dict** (*dict, optional*) – Dictionary returned by the rivus.io.plot.fig3d function.

• **profiler** (*pandas.Series, optional*) – Series containing profiled process name and execution time pairs. Execution time is measured in seconds

Returns **int** – run_id of the initialized run row in the DB.

**purge_run**( **engine**, **run_id**)  
Delete all rows related to run_id across all tables. **This is not a throughout reliable function, and can do some harm. Use it with caution, and at your own risk!**

Parameters

• **engine** (*sqlalchemy engine whit psycopg2 driver*) – For managing connection to the DB.

• **run_id** (*int*) – run_id of the initialized run row in the DB. Used to identify related data to be removed: directly (table has run_in as FK) and indirectly (table has FK of an Entity with run_id FK)

Returns **None**

**store**( **engine**, **prob**, **run_id=None**, **graph_results=None**, **run_data=None**, **time_series=None**, **constants=None**)  
Store I/O plus extras of a rivus model into a postgres DB.

Parameters

• **engine** (*sqlalchemy engine whit psycopg2 driver*) – For managing connection to the DB.

• **prob** (*pyomo ConcreteModel*) – Created by rivus.create_model()

• **run_id** (*int, optional*) – run_id of an initialized run row in the DB. If omitted: init_run() will be called with run_data.

• **graph_results** (*iterable, optional*) – Results of the graph analysis. Each graph should have its own dict. For implemented result keys see _handle_graph. E.g. [{‘is_connected’:True, ‘is_minimal’:True}, {‘is_connected’:True}]

• **run_data** (*dict, optional*) – Keyword arguments to be passed to init_run(). runner, start_ts, status, outcome, comment, plot_dict, profiler

• **time_series** (*None, optional*) – TODO If already present at function call, this could save time.
• **constants** *(None, optional)* – TODO If already present at function call, this could save time.

**Returns** None

**Raises** Exception caught during data export.

### 4.4.2 rivus.plot

Interactive 3D data visualization! See live example [here](#) (May take a while to load.)

Plotly has a great [documentation](#) and an extensive examples library.

It is available also available as a JavaScript library, which can help when a web-app GUI is added to rivus. (Client-side visualization.)

To ensure interoperability, it thinks in dictionaries (JSONs). This gives a greater flexibility then e.g. the API of matplotlib.

Adding extra hover-over information (line length, capacity, you-name-it) is also highly flexible.

The structure can be explained easily:

```python
data = cap_layers + hub_layer + markers
fig = dict(data=data, layout=layout)
```

where **cap_layers**, **hub_layer** and **markers** simply lists of dictionaries are, which dictionaries have the key-value pairs of the Scatter3d structure (class).

We use raw dicts instead of Plotly’s “classes” because they are mainly the same, but much faster. Still, Loading the plot can take a while if there are a lot of elements to render. (For each edge we have a scatter3d dict in the data list, which are grouped together by the same legendgroup key value.)

The plot consists of layers, stacked upon each other. Each of these represent a commodity. All of the edges are shown in each layer, but if in one no carrier was built, it is displayed dashed. If the edge is stroked through, some amount of capacity was built out there. The width of the edges are in proportion with that amount.

Diamond shapes represent the sources in the vertices.

Vertical lines show processes (commodity conversions).

All of the elements can be toggled with the help of the menu in the upper right corner.

---

**Note:** As the legends (with which you can turn off/on the layers) are generated from the first elements per legend-groups in the data list, there are now dummies as first ones to ensure a nicer look.

```python
fig3d(prob, comms=None, linescale=1.0, use_hubs=False, hub_opac=0.55, dz=5, layout=None, verbose=False)
```

Generate 3D representation of the rivus results using plotly

**Parameters**

- **prob** *(rivus_archive)* – A rivus model (later extract of it)
- **comms** *(None, optional)* – list/ndarray of commodity names to plot, Order: [‘C1’, ‘C2’, ‘C3’] -> Bottom: C1, Top: C3
• linescale *(float, optional)* – A multiplier to get proportionally thicker lines.

• use_hubs *(bool, optional)* – Switch to depict hub processes.

• hub_opac *(float, optional)* – 0-1 opacity param.

• dz *(number, optional)* – Distance between layers along ‘z’ axis.

• layout *(None, optional)* – A plotly layout dict to overwrite default.

• verbose *(bool, optional)* – To print out progress and the time it took.

**Example**

```python
import plotly.offline as po
fig = fig3d(prob, ['Gas', 'Heat', 'Elec'], hub_opac=0.55, linescale=7)
# for static image
# po.plot(fig, filename='plotly-game.html', image='png')
pot.plot(fig, filename='plotly-game.html')
```

**Returns** *plotly compatible figure* *dict* (in plotly everything is kinda a dict.)*

**Note:** Greatly inspired by Example1 and Example2.

### 4.5 rivus.utils

#### 4.5.1 prerun

Collection of small rivus related helper functions

**In use to avoid multiple solutions of the same task, like:**

• Setting up the solver.

• Todo: Create needed directories

**setup_solver** *(optim, logfile='solver.log', guro_time_lim=12000, guro_mip_focus=2, guro_mip_gap=0.001, guro_threads=None, log_to_console=True)*

Change solver options to custom values.

**Parameters**

• *optim* – (pyomo Solver object): See usage for example

• *logfile* *(str, optional)* – default='solver.log’ Name (Path) to the logfile

• *guro_time_lim* *(int, optional)* – unit is seconds | default=12000

• *guro_mip_focus* *(int, optional)* – default=2 1=feasible, 2=optimal, 3=bound
• **guro_mip_gap** *(float, optional)* – our default=.001 (gurobi’s default: 1e-4)

• **guro_threads** *(None, optional)* – parallel solver tasks | default=None If None, no Threads parameter is set
  
  (gurobi takes <=CPU_count threads automatically)

  If greater than CPU_count then it is threshold to CPU_count If less than CPU_count then Thread is set with the parameter

• **log_to_console** *(Boolean, optional)* – is not piped to the stdout.

Usage: optim = SolverFactory('glpk') optim = setup_solver(optim, logfile=log_filename)

### 4.5.2 notify

**email_me** *(message, sender, send_pass, recipient, smtp_addr, smtp_port, subject='[rivus][notification]')*

Send notification message through email server.

Parameters

- **message** *(str)* – Body of the e-mail
- **sender** *(str)* – The e-mail account through which the email will be sent.
  E.g. tum.robot@gmail.com
- **send_pass** *(str)* – Password of sender. Hopefully read from a file, which is not added to Git...
- **recipient** *(str)* – The e-mail account of you, where you want to get the notification.
- **smtp_addr** *(str)* – SMTP Address. Like “smtp.gmail.com”
- **smtp_port** *(int)* – SMTP Port. Like 587
- **subject** *(str, optional)* – The subject of the mail...

Returns integer – 0 - if run through without exception -1 - if encountered with a problem (mainly for unittest)

### 4.5.3 runmany

Functions are collected here, which can be useful in case of a massive runs involving analysis of a broader parameter-space.

**parameter_range** *(data_df, index, column, lim_lo=None, lim_up=None, step=None, zero_root=None)*

Yield values of the parameter in a given range :param data_df: Original data frame, where the target parameter can be found. :type data_df: DataFrame :param index: DataFrame .loc parameter to locate the parameter value.

E.g.: [‘Gas power plant’, ‘CO2’, ‘Out’] or ‘Gas’

Parameters
• **column** *(str)* – Label of the column, where the parameter is. E.g.: ‘ratio’ or ‘cap-max’

• **lim_lo** *(None, optional)* – Proportional parameter. If omitted, 90% of the original.

• **lim_up** *(None, optional)* – Proportional parameter. If omitted 110% of the original.

• **step** *(None, optional)* – Proportional parameter. The difference between two following yielded values.

• **zero_root** *(None, optional)* – If the selected parameter is 0, then the default method using proportions will fail. Use this value to set the root for the parameter range.

**Returns** *DataFrame* – A modified version of xls[df_name]

### 4.5.4 pandashp

pandashp: read/write shapefiles to/from special DataFrames

Offers two functions read_shp and write_shp that convert ESRI shapefiles to pandas DataFrames that can be manipulated at will and then written back to shapefiles. Opens up data manipulation capabilities beyond a simple GIS field calculator.

**Usage:**

```python
import pandashp as pdshp  # calculate population density from shapefile of cities (stupid, I know)
cities = pdshp.read_shp('cities_germany_projected')
cities[popdens'] = cities['population'] / cities['area']
pdshp.write_shp(cities, 'cities_germany_projected_popdens')
```

• **bounds** *(df)*

  Return a DataFrame of minx, miny, maxx, maxy of each geometry.

• **find_closest_edge** *(polygons, edges, to_attr='index', column='nearest')*

  Find closest edge for centroid of polygons.

  **Parameters**

  • **polygons** – a pandas DataFrame with geometry column of Polygons

  • **edges** – a pandas DataFrame with geometry column of LineStrings

  • **to_attr** – a column name in DataFrame edges (default: index)

  • **column** – a column name to be added/overwrite in DataFrame polygons with the value of column to_attr from the nearest edge in edges

  **Returns** a list of LineStrings connecting polygons’ centroids with the nearest point in edges. Side effect: polygons recieves new column with the attribute value of nearest edge. Warning: if column exists, it is overwritten.

• **match_vertices_and_edges** *(vertices, edges, vertex_cols=('Vertex1', 'Vertex2'))*

  Adds unique IDs to vertices and corresponding edges.

  Identifies, which nodes coincide with the endpoints of edges and creates matching IDs for matching points, thus creating a node-edge graph whose edges are encoded purely by node ID pairs. The optional argument vertex_cols specifies which DataFrame columns of edges are added, default is ‘Vertex1’ and ‘Vertex2’.

  **Parameters**
• **vertices** – pandas DataFrame with geometry column of type Point
• **edges** – pandas DataFrame with geometry column of type LineString
• **vertex_cols** – tuple of 2 strings for the IDs numbers

**Returns** Nothing, the matching IDs are added to the columns vertex_cols in argument edges

**read_shp** *(filename)*
Read shapefile to dataframe w/ geometry.

**Parameters** **filename** – ESRI shapefile name to be read (without .shp extension)

**Returns** pandas DataFrame with column geometry, containing individual shapely Geometry objects (i.e. Point, LineString, Polygon) depending on the shapefiles original shape type

**total_bounds** *(df)*
Return bounding box (minx, miny, maxx, maxy) of all geometries.

**write_shp** *(filename, dataframe, write_index=True)*
Write dataframe w/ geometry to shapefile.

**Parameters**
• **filename** – ESRI shapefile name to be written (without .shp extension)
• **dataframe** – a pandas DataFrame with column geometry and homogenous shape types (Point, LineString, or Polygon)
• **write_index** – add index as column to attribute tabel (default: true)

**Returns** Nothing.

### 4.6 rivus.test

Write tests to every function where there are clearly definable input(s) and output(s).

Enlighten the main logic of the test here if it is not trivial.

#### 4.6.1 test_db

**class RivusDBTest**(methodName='runTest')

**test_df_insert_query** ()
Are the stored dataframes and the retrieved ones identical?

• Comparison form of frames is after create_model. (index is set)
• Comparison form expects that input dataframes only have meaningful columns. (See pull request #23)
• Only implemented dataframes are tested.

**Note:** Requires a config.json file in the root of rivus-repo with the database credentials.
For Example:
4.6.2 test_gridder

```python
class RivusGridTest (methodName='runTest')
```

4.6.3 test_main

```python
class RivusMainTest (methodName='runTest')
```

4.6.4 test_utils

```python
class RivusUtilsTest (methodName='runTest')
```

test_email_notification()

It only can test, whether the notification function run through successfully.

Can be useful for quick testing the email parameters in the config file

**Note:** Requires a `config.json` file in the root of rivus-repo with the database credentials. For Example:

```json
{
    "email" : {
        "s_user" : "robot.mail@gmail.com",
        "s_pass" : "TheAnswerIs42!",
        "r_user" : "my.mail@gmail.com",
        "smtp_addr" : "smtp.gmail.com",
        "smtp_port" : "587"
    }
}
```

test_parameter_range()

Load minimal example data and test with one of the parameters

- Tests for side-effects on the original.
- Tests for the awaited number of parameters.
- Tests for range of the parameters
CHAPTER 5

Reference - Mathematical

Todo: Extract description from ojdo’s thesis

Now this is more or less a placeholder.
## 5.1 Model Variables - Meaning

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SETS</strong></td>
<td></td>
</tr>
<tr>
<td>Commodity</td>
<td></td>
</tr>
<tr>
<td>commodity</td>
<td>Commodities</td>
</tr>
<tr>
<td>co_demand</td>
<td>Commodities that have demand in edges</td>
</tr>
<tr>
<td>co_source</td>
<td>Commodities that may have a source at some vertex/vertices</td>
</tr>
<tr>
<td>co_transportable</td>
<td>Commodities that may be transported through edges</td>
</tr>
<tr>
<td>co_allowed_max</td>
<td>Commodities that have a maximum allowed generation (e.g. CO2)</td>
</tr>
<tr>
<td><strong>Prozess</strong></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>Processes, converting commodities in vertices</td>
</tr>
<tr>
<td>process_input_tuples</td>
<td>Commodities consumed by processes</td>
</tr>
<tr>
<td>process_output_tuples</td>
<td>Commodities emitted by processes</td>
</tr>
<tr>
<td><strong>Hub</strong></td>
<td></td>
</tr>
<tr>
<td>hub</td>
<td>Hub processes, converting commodities in edges</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>Timesteps</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td></td>
</tr>
<tr>
<td>storage</td>
<td>Connection points between edges, for source and processes</td>
</tr>
<tr>
<td><strong>Graph</strong></td>
<td></td>
</tr>
<tr>
<td>edge</td>
<td>Undirected street segments, for demand and hubs</td>
</tr>
<tr>
<td>arc</td>
<td>Directed street segments, for power flows</td>
</tr>
<tr>
<td><strong>Costs</strong></td>
<td></td>
</tr>
<tr>
<td>cost_type</td>
<td>supply (kW of commodity in edge at time)</td>
</tr>
<tr>
<td><strong>VARIABLES</strong></td>
<td></td>
</tr>
<tr>
<td>Edges and Arcs</td>
<td></td>
</tr>
<tr>
<td>Pin</td>
<td>power flow (kW of commodity into arc at time)</td>
</tr>
<tr>
<td>Pot</td>
<td>power flow (kW of commodity out of arc at time)</td>
</tr>
</tbody>
</table>

## 5.2 Map: Greek-English

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Doc-SourceCode</th>
<th>Doc-PhD</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho$</td>
<td>source?</td>
<td>commodity is introduced</td>
<td>Commodity source power flow</td>
</tr>
<tr>
<td>$P_{in}, P_{ot}, P_{max}$</td>
<td>Pmax</td>
<td>commodity is transported</td>
<td>Power</td>
</tr>
<tr>
<td>$\kappa_{eh,vp}$</td>
<td>Kappa_</td>
<td>commodity is consumed</td>
<td>Capacity (Hub, ProcessThroughput)</td>
</tr>
<tr>
<td>$\epsilon_{eht}$</td>
<td>hub_io?</td>
<td>commodity is converted</td>
<td>Hub proc activity</td>
</tr>
<tr>
<td>$\tau_{vpt}$</td>
<td>proc_tau</td>
<td>X</td>
<td>Process power throughput</td>
</tr>
<tr>
<td>$\epsilon_{v,pct}$</td>
<td>proc_io?</td>
<td>flows?</td>
<td>Process power flows</td>
</tr>
<tr>
<td>$\sigma_{peak}$</td>
<td>consume?</td>
<td>commodity is consumed</td>
<td>Supply in edge at time</td>
</tr>
</tbody>
</table>
CHAPTER 6

Contribution

6.1 Code Style

6.1.1 Naming conversions

Still not written in stone but the snake_case seems to dominate the code base. If you are new to the project, take up that one.

Most importantly, use descriptive names and stick to one style and use it consequently.

Regarding functions, variables, class etc. we follow the good advice of PEP8.

**Note:** Functions: Please use an underscore like `_spam_and_helps()` for functions which should not be used outside of the sub-package (package). Start function names without an underscore if they are meant to be used outside of the package/sub-package. E.g. `flying_circus()`.

For the rest, it is still to be defined. But we basically follow PEP8, but do not forget how PEP8 itself starts:

“A Foolish Consistency is the Hobgoblin of Little Minds”

So use it when it makes sense, and do not where it does not. We would even suggest looking for an auto-pep8 plug-in for your favourite IDE or text editor. Or go bare bones and get something like flake8 to check your code. If it is too annoying, you can always add some guidelines to be ignored. But at the end of the day, it will help you and us to read the code.

**Todo:** Look into `rivus.main.rivus` and reformat code where it makes sense. (Mathematical notation vs. descriptiveness)
6.2 Unitests

**Note:** Unit tests are criteria for pull-request acceptance.

As several bugs could have been avoided with unit tests, we want to embrace testing in the development workflow.

You can find and extend the tests in rivus-repo/rivus/tests/test_*.py Just add your own test_subpackage.py and start with the following skeleton:

```python
import unittest
# import what you want to test with an absolute path.
# If I want to test line_length from main.rivus:
from rivus.main.rivus import line_length

# give a nice class name.
class RivusMainTest(unittest.TestCase):
    def test_line_length(self):
        # here comes the test logic
        self.assertTrue(4/2 == 4//2, msg='What? Is not that always true???

pass
```

To run all tests:

```bash
>>> cd /to/rivus/repo/
>>> python -m unittest
Ran 10 tests in 4.572s
OK
```

To run specific test(s):

```bash
>>> cd /to/rivus/repo/
>>> python -m unittest rivus.tests.test_db
Ran 1 test in 1.213s
OK
```

6.3 Profiling

Although the whole project is not yet in a performance oriented phase, it can be very helpful in the long run if more and more profiling is involved within the contributions.

For decision between short expression: use `timeit.timeit()`. E.g. is `.at[]` or `.loc[]` indexer slower from a DataFrame?

```bash
>>> import timeit.timeit as tit
>>> dfout = m.r_out.to_frame()  # m is a rivus problem instance
```
To detect overall (function-level) hotspots (where the most time is spent) you can use cProfile with pstats. See blog post

```python
python -m cProfile -o runme.profile runme.py  # Execute with profiler
python -m pstats runme.profile  # Interactive analyser
% sort cumulative
% stats 10
```

To go deeper, you can use the jupyter magic, `%lprun` (line_profiler). See end of tutorial and other

```bash
%load_ext line_profiler
%lprun -f slow_functions.main slow_functions.main()
```

---

**Note:** Profiling is encouraged before pull-request.

### 6.4 Documentation

Nobody can explain better what your code does than you. The doc-strings are essential, and I would not suppose anybody would submit code without it ;)

Please take the time and jump into the conventions of RTD (Readthedocs) the following short description should be enough to get you started and ensure the success of your contribution.

---

**Note:** We use nepoleon (autodoc) extensions to parse the doc-strings. You can decide whether you choose NumPy or Google style.

Google it, there is a good chance that there is already a plug-in for your favourite IDE or text editor. (Yes, there is even one for vim...)

The whole RtD (Sphinx) workflow builds on RST (reStructuredText). So format your Examples, Notes, References in the doc-strings also with rST syntax.

Doc-string template:

```python
"""Summary line.

Extended description of function.

Parameters
----------
arg1 : int
   Description of arg1
arg2 : str
   Description of arg2

Returns
-------
"""
```
bool
    Description of return value

Example
-------
::
    this_will = 'be formatted as nice code!'

Note
----
+ This is a bullet list
+ E.g. for limitations...
.. note::
    This will draw attention to the content.
.. warning::
    This is for deprecate warnings and such.

Note: For autodoc/napoleon to work, readthedocs must be able to build the whole project. It will create
a new environment and install the packages from environment.yml. This is told to readthedocs through
the readthedocs.yaml config file. See project structure

The whole process of the documentation is depicted in the following figure.

How to write (and build) documentation locally:

- Install Sphinx
  
  pip install sphinx sphinx-autodoc

- Install RtD local theme
  
  pip install sphinx_rtd_theme

- Build the docs manually into doc/_build/html/
  
  cd /rivus/repo/doc
  make html

- Or start autobuild, which will detect changes and autobuild the new html.
  
  cd /rivus/repo
  sphinx-autobuild doc/ doc/_build/html

Tips:

- online table editor

- online pandoc converter It is an anything to anything converter. Here: (HTML, Markdown, …) -> rST
Fig. 6.1: How to write docs. Dashed: suggested optional step. After commit, the rest should be done automatically.

- online rST/Sphinx editor
- You can also try Atom and its rST packages.

**Note:** Documentation is a criteria for pull-request acceptance.

### 6.5 Doc TODOs

Summary of the ToDos from the *whole* documentation.

**Todo:** Look into `rivus.main.rivus` and reformat code where it makes sense. (Mathematical notation vs. descriptiveness)

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/rivus/checkouts/latest/doc/contribution.rst`, line 33.)

**Todo:** Set version number to 0.2 after accepting the PRs

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/rivus/checkouts/latest/doc/index.rst`, line 81.)
Todo: Extract description from ojdo’s thesis

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/rivus/checkouts/latest/doc/reference_math.rst, line 5.)
Short Introduction

• **rivus** is a MILP model for multi-commodity energy infrastructure networks systems with a focus on high spatial resolution.

• It finds the minimum cost energy infrastructure networks to satisfy a given energy distribution for possibly multiple commodities (e.g. electricity, heating, cooling, . . .).

• Time is represented by a (small) set of weighted time steps that represent peak or typical loads.

• Spatial data can be provided in form of shapefiles, while technical parameters can be edited in a spreadsheet.

• Sister project to urbs which has a focus on high time resolution analysis.
Besides the numerical results (obtainable programmatically or as an excel report), rivus is can visualize the results. You can see a unified figure of the typical result plots below created by `rivus.main.rivus.result_figures()`, which in turn is a wrapper around `rivus.main.rivus.plot()`. Merge was done only for better spacing in the documentation.

Unified rivus capacity output of a smaller city.

**Yellow:** Electricity network capacities

**Red:** Heat network capacities

**Brown:** Gas network capacities

**Gray:** Longitude and latitude guidelines.

**Symbols:** Diamond shapes represent energy sources. The width of the lines represent the amount of built capacity. Triangles represent energy conversion processes. Pointing upwards indicate generation, downwards indicates consumption of that commodity.

If got curious go to *Installation*
CHAPTER 9

Change-log

---

**Todo:** Set version number to 0.2 after accepting the PRs

---

### 9.1 Version 0.2

- 2017-10
- Major bug-fix regarding line length calculation (#13)
- Major bug-fix regarding usage of hubs in vertices (#37)
- Restructuring (Issue5, PR8)
- 3D visualization for results (PR13)
- Abstract street network generator (PR8, PR28)
- 3D interactive visualization for results (#17)
- Graph/Network analysis sub package (#22)
- PostgreSQL+PostGIS integration (#24)
- Unittests (#33)
- Line length calc bugfix
- Save function extension (#16)
- Excel parsing (#23)
- Speed-ups (#27)
- Fix Warnings (#30)
9.2 Version 0.1

- 2015-05
- Initial release.
Dependencies

• Python version 3.x is supported. Version 2.7 may work.

• Pyomo for model equations and as the interface to optimisation solvers (CPLEX, GLPK, Gurobi, . . .). Version 4 recommended, as version 3 support (a.k.a. as coopr.pyomo) will be dropped soon.
  – ConcreteModel is the basis to formulate our optimisation problem.

• matplotlib for plotting

• Plotly for interactive 3D plots [extension]
  – We use the offline variants of this package.
  – Also compatible with the jupyter notebook.

• Pandas for input and result data handling, report generation
  – Pandas’ DataFrame is used very extensively throughout rivus.

• SQLAlchemy and psycopg2 for database integration [extension]
  – Pandas’s DataFrame works together with SQLAlchemy.

• NetworkX for network analysis [extension]
Python Module Index

r
rivus.graph.analysis, 28
rivus.graph.to_graph, 27
rivus.gridder.create_grid, 29
rivus.gridder.extend_grid, 31
rivus.io.db, 32
rivus.io.plot, 35
rivus.main.rivus, 24
rivus.tests.test_db, 39
rivus.tests.test_gridder, 40
rivus.tests.test_main, 40
rivus.tests.test_utils, 40
rivus.utils.notify, 37
rivus.utils.pandashp, 38
rivus.utils.prerun, 36
rivus.utils.runmany, 37
Index

B
bounds() (in module rivus.utils.pandashp), 38

C
create_model() (in module rivus.main.rivus), 22
create_square_grid() (in module rivus.gridder.create_grid), 29

D
df_from_table() (in module rivus.io.db), 32

E
e-mail_me() (in module rivus.utils.notify), 37
extend_edge_data() (in module rivus.gridder.extend_grid), 31

F
fig3d() (in module rivus.io.plot), 35
find_closest_edge() (in module rivus.utils.pandashp), 38

G
get_constants() (in module rivus.main.rivus), 25
get_entities() (in module rivus.main.rivus), 25
get_entity() (in module rivus.main.rivus), 24
get_onset_names() (in module rivus.main.rivus), 25
g-get_source_candidates() (in module rivus.gridder.create_grid), 30
get_timeseries() (in module rivus.main.rivus), 26

I
init_run() (in module rivus.io.db), 33

L
line_length() (in module rivus.main.rivus), 22
list_entities() (in module rivus.main.rivus), 25
load() (in module rivus.main.rivus), 24

M
match_vertices_and_edges() (in module rivus.utils.pandashp), 38
minimal_graph_anal() (in module rivus.graph.analysis), 28

P
parameter_range() (in module rivus.utils.runmany), 37
plot() (in module rivus.main.rivus), 23
purge_run() (in module rivus.io.db), 34

R
read_excel() (in module rivus.main.rivus), 23
read_shp() (in module rivus.utils.pandashp), 39
report() (in module rivus.main.rivus), 24
result_figures() (in module rivus.main.rivus), 23
rivus.graph.analysis (module), 28
rivus.graph.to_graph (module), 27
rivus.gridder.create_grid (module), 29
rivus.gridder.extend_grid (module), 31
rivus.io.db (module), 32
rivus.io.plot (module), 35
rivus.main.rivus (module), 22–24
rivus.tests.test_db (module), 39
rivus.tests.test_gridder (module), 40
rivus.tests.test_main (module), 40
rivus.utils.notify (module), 37
rivus.utils.pandashp (module), 38
rivus.utils.prerun (module), 36
rivus.utils.runmany (module), 37
RivusDBTest (class in rivus.tests.test_db), 39
RivusGridTest (class in rivus.tests.test_gridder), 40
RivusMainTest (class in rivus.tests.test_main), 40
RivusUtilsTest (class in rivus.tests.test_utils), 40
S
save() (in module rivus.main.rivus), 24
save_log() (in module rivus.main.rivus), 24
setup_solver() (in module rivus.utils.prerun), 36
store() (in module rivus.io.db), 34

T
test_df_insert_query() (RivusDBTest method), 39
test_email_notification() (RivusUtilsTest method), 40
test_parameter_range() (RivusUtilsTest method), 40
to_igraph() (in module rivus.graph.to_graph), 27
to_nx() (in module rivus.graph.to_graph), 27
total_bounds() (in module rivus.utils.pandashp), 39

V
vert_init Commodities() (in module rivus.gridder.extend_grid), 31

W
write_shp() (in module rivus.utils.pandashp), 39