**Academic Year**

2024 - 2025

**Software engineering for real-time electricity market bidding**

Sameer Baruwal

Bachelor’s thesis to obtain the degree of Bachelor of Science in

**Electronics and ICT Engineering Technology**

Promotor

Benoit De Vrieze

Copromotor

Prof. dr. Peter Hellinckx

Inhoud

[1 Abstract 3](#_Toc198638567)

[2 Introduction 3](#_Toc198638568)

[3 Background 4](#_Toc198638569)

[3.1 The electricity market 4](#_Toc198638570)

[3.2 The ASSUME Framework 4](#_Toc198638571)

[3.3 Big O 5](#_Toc198638572)

[3.3.1 Math 5](#_Toc198638573)

[4 Methodology 7](#_Toc198638574)

[4.1 Initial simulation 7](#_Toc198638575)

[4.2 Profiling 8](#_Toc198638576)

[4.2.1 cProfile & yappi 8](#_Toc198638577)

[4.2.2 Tracemalloc & memory\_profiler 9](#_Toc198638578)

[5 Results 9](#_Toc198638579)

[5.1 Simulation results 9](#_Toc198638580)

[5.2 Profiling 10](#_Toc198638581)

[5.2.1 Time 11](#_Toc198638582)

[5.2.2 Memory 14](#_Toc198638583)

[5.3 Optimization 14](#_Toc198638584)

[5.3.1 Aggregate\_step\_amount function 14](#_Toc198638585)

[5.3.2 Calculate\_cashflow function 15](#_Toc198638586)

[6 Conclusion 16](#_Toc198638587)

[7 Acknowledgements 16](#_Toc198638588)

[8 Appendices 16](#_Toc198638589)

[9 References 18](#_Toc198638590)

# Abstract

This study’s main goal is to simulate ten thousand residential agents in five minutes on a model of the Belgian electricity market.

# Introduction

This research delves into the ASSUME framework to simulate an energy market with ten thousand residential agents. It was chosen so that there was no need to develop a market from scratch, which can be quite complex. The main goal of this project is to simulate the yearly market operations of ten thousand residential agents in five minutes on a model of the Belgian electricity market. To do this, we need first create and simulate a scenario that approximates the market, analyse its performance by means of profiling, and then optimize the framework efficiently to decrease its simulation time. The research questions are:

* What is the processing time and memory consumption of the initial scenario?
* What are the key time and memory bottlenecks?
* How can we optimize the ASSUME framework efficiently to simulate ten thousand agents in five minutes?

This research displays a future scenario where residential consumers, instead of buying electricity from an electricity supplier, can buy it directly off the market. This can be done using a sort of aggregator… **which is?** Like an AI machine that buys electricity at a low cost. This is also the purpose of this research, to simulate what it is like when residents buy electricity directly from the market. This helps us understand the scenario before a real-life implementation can be realized. It links with modelling for sustainability… **why is it sustainable? Need to get back to this**

The next section will discuss the literature background to this research project. It will discuss the working mechanism of the electricity market and how the ASSUME framework works both internally and externally to model the market. The Big O notation, its mathematical foundation and how it can be used to theoretically profile a function will also be discussed in the last paragraph of the section. The section that follows discusses the ways that the project was set up. This includes the initial set up of a simulation environment that models the Belgian market, the analysis of the time and memory bottlenecks with various profiling techniques and the optimization methods that were considered to overcome the bottlenecks. The results section will be graphical section where the results of the simulation, profiling, optimization and a comparison between the before and after will be discussed. Finally, we conclude this paper by summarizing the project, answering the research question, discussing the limitations and potential improvements.

# Background

## The electricity market

The electricity market is very different from the publicly known markets such as the stock market. The main difference lies in the nature of the commodity being traded and how it is used. The electricity market trades electricity, which is an instantaneous and non-storable commodity while the stock market trades ownership of companies in the form of shares.

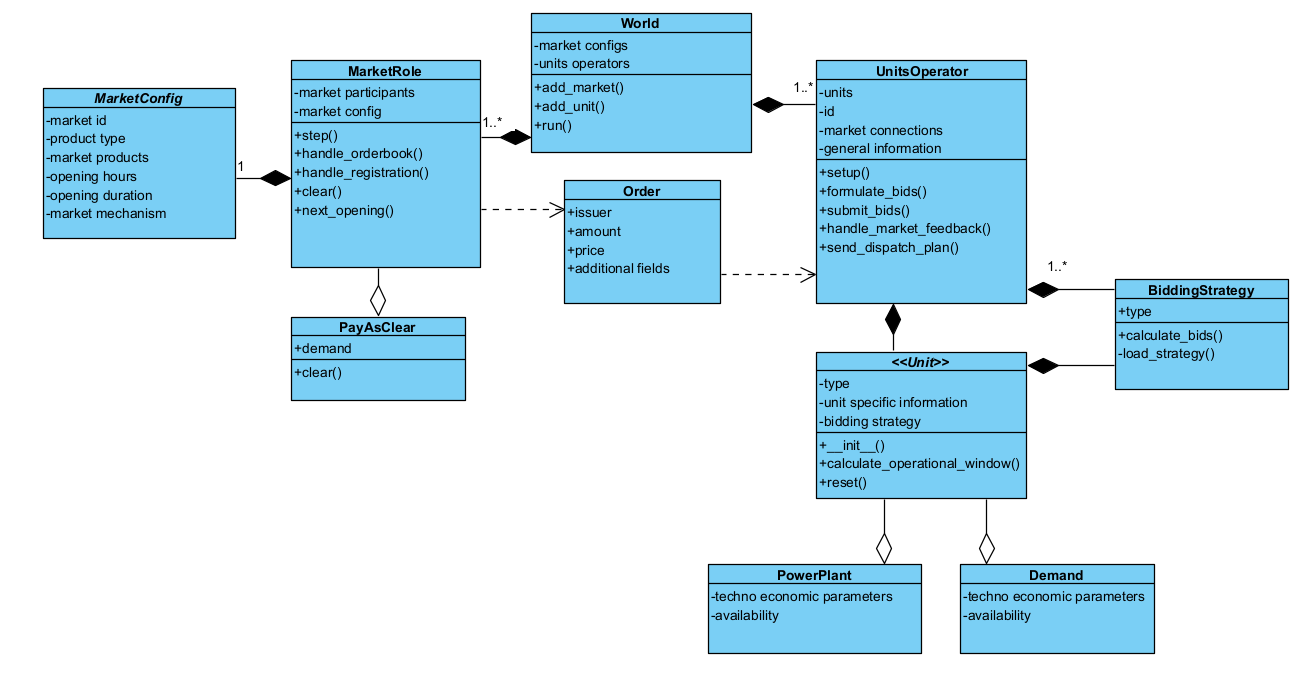
The electricity market works as follows. The members of the market, producers and consumers, submit orders for buying and/or selling power, which are registered in an orderbook. These orders reflect the demand and supply of the market at a specific moment in time. Based on the orderbook, a market clearing price, MCP, is calculated using the merit order principle. This principle guarantees the lowest possible prices to satisfy demand on the power market, as the generation with the lowest costs (or the willingness to sell at the lowest price) is dispatched first. The most expensive unit that must be activated to meet the demand sets the price. The least expensive units are usually renewables like wind and solar while the most expensive are gas and coal plants.

The members usually operate in two markets, the Day-Ahead and the Intraday market, that fulfil their own purpose. The day ahead market operates through a blind auction that takes place once a day, all year around where all the hours of the following day are traded. The Intraday market offers the possibility to trade even more in the short term. On the Intraday continuous market, participants trade 24 hours a day, with delivery on the same day. As soon as a buy- and sell-orders match, the trade is executed. Electricity can be traded for up to 5 minutes before delivery and through hourly, half-hourly or quarter-hourly contracts. As this allows for a high level of flexibility, members use the Intraday market to make last-minute adjustments and to balance their positions closer to real time.

Once the trade is complete, the transaction is cleared and settled. Clearing ensures the proper fulfilment of each contract registered on the market. The clearing entity steps in and becomes the contractual partner for both the buyer and the seller to ensure the fulfilment of each trade and to mitigate counterparty risk. By matching supply and demand, the market ensures transparent and reliable prices, and the market operators make sure the electricity is delivered and paid for. [1]

## The ASSUME Framework

The ASSUME framework in an open-source toolbox for agent-based simulation of European electricity markets. ASSUME in general intended for everyone searching to understand market dynamics of energy markets. The framework provided a wide range of possibilities to tailor to the user’s case. The starting point is the World entity where you can define markets and units to closely simulate a real-world example. The architecture of the framework is depicted in the following figure. It can be roughly divided into two parts. On the left side of the world class are the markets located and on the right side the market participants, which are here named units. Both are connected via the orders that market participants place on the markets.

*Figure 1: Architecture of the ASSUME framework without reinforcement learning*

It is also crucial to understand how the framework works under the hood because it defines the methodology that we use and the results that are shown. ASSUME uses the Mango agent framework for the agents messaging and Python’s asynciofor asynchronous execution. Mango supplies the abstraction like *RoleAgent, Role* and *Container* which manage agents, their roles and how they communicate. ASSUME instantiates a container, creates agents that each have roles like *UnitsOperator* and *MarketRole,* and registers them in the container. When the simulation is run, the World class calls the asynchronous run function that activates the container. The agents activate their roles like *UnitsOperator* that in turn activate the units that it manages. The agents use asyncio’s *async* and *await* functions to perform non-blocking, concurrent tasks. They do this by yielding control and wating for a message when it comes across an *await* function and resuming the task when a message is received. This way, a single agent does not block the execution of the rest of the code and the agents appear to work concurrently.

## Big O

An understanding of the Big O notation and its mathematical foundation is essential as it will be used throughout this paper when analysing the computational performance of the algorithms. The big O notation is a way of describing how fast an algorithm grows as the size of the input grows. It is not an indication of how long something takes, but of how the performance will scale, especially in the worst-case scenario.

### Math

#### Definition

# Methodology

**Mention the hardware at the start**

The work is split into three sequential steps: initial simulation, profiling and optimization. These three steps will be explained in depth in the following paragraphs

## Initial simulation

The task here is to approximate the Belgian electricity market. We approximate because building an exact model is not the core of the research. That is why we initialize 5 powerplants: nuclear, fossil fuels, wind, biofuel and solar that represent about 95 % of the power generation in Belgium. The five units are set up with their maximum power output and cost which is fixed. The data is gathered from the International Energy Agency (IEA) Data Services [1]. Using this data, we can calculate the average hourly generation which is shown in column four.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Yearly (GWh) | Percentage | Average Hourly (MWh) |
| Nuclear | 43879 | 0.493166543 | 5009 |
| Fossil fuel | 21941 | 0.24660013 | 2505 |
| Wind | 12352 | 0.138827073 | 1410 |
| Solar | 6876 | 0.077281003 | 785 |
| Biofuel | 3926 | 0.04412525 | 448 |
| Total | 88974 | 1 | 10157 |

*Figure 3: The total and hourly generation per type of Belgium 2022*

To simulate ten thousand agents, we use the meter data gathered from the Fluvius database [2]. Fluvius is the distribution service operator (DSO) of Flanders. The database consists of 1300 meter profiles with two columns, one representing the consumption and the other representing production. The amount of meter profiles is not sufficient however to simulate ten thousand agents, so a random meter profile is selected every time a unit is created. The net load, the difference between the consumption and production, is the residents demand. A negative demand means that he can sell this excess power on the market. This scenario does not approximate the Belgian market, because the electricity consumption is much higher. To take the rest of the market into account, an Agent0 is created with its demand as the measured total load on the Belgian grid, gathered from the Elia database [3]. The set up of the demand and supply units is an abstraction used only for simulation purposes. In a real life scenario, consumers would not trade full demand at a single moment but rather have a baseline contract and trade the amount around the baseline, first in coarse blocks and then finer as the moment of delivery comes closer. Producers on the other hand do not produce the same amount of power every hour. Their supply is dependent on various factors such as demand, availability of product and time of day.

The ASSUME framework provides the possibility of using csv files to defines units, the market is defined using a yaml file. We store all these files in a certain folder and give the path name and the world entity as parameters into the simulation function of the framework. The output is saved in an SQLite database at the path defines by the user, given as a parameter when initializing the World entity. The units are defined above. The market is defined for the year 2022 because this is the most recent year for which the Fluvius meter data is available. The market is a Day-Ahead market.

The results of the performance analysis and the optimization was done using 10 agents, nine residential and the agent0. The reasoning is that the simulation runtime takes more time for more agents as will be shown in the results section. Ten agents show the bottlenecks well enough and simulating with more agents would just be time lost, wating for it to finish. This means that the results that we obtain for simulating ten thousands agent are by means of extrapolation. **Maybe add something**

## Profiling

It needs to be made clear that the research is rather focused on the time profiling than the memory profiling since it is a bigger performance issue as will be shown in the results section. A modern pc with RAM of at least 8 GB is sufficient to simulate ten thousand agents using the ASSUME framework memory wise.

The profiling was done using the following time and memory profilers.

* cProfile & yappi (time)
* tracemalloc & memory\_profiler (memory)

Each profiler will be explained with reasoning on why it was or was not chosen to do the final profiling.

### cProfile & yappi

There are two types of profilers: statistical and deterministic. Statistical profilers periodically check what the program is doing by sampling the call stack, they produce low overhead and will not slow down the program much, but they are not very precise in terms of timing and call count. Deterministic profilers on the other hand are very precise because they log every single function call and return, but this also means that they produce a lot of overhead. A statistical profiler would serve no purpose because the precision of the timing and call count is very important in this research, so we chose deterministic profilers: cProfile and Yappi (Yet Another Python profiler).

cProfile [4] is a built-in python module while yappi [5] is a python package that must be installed first. The main difference is that yappi support multithreaded, asyncio and gevent profiling. It can also track either WALL time or CPU time while cProfile only tracks CPU time. The ASSUME framework is built on top of the Mango framework which in turn is built on top of asyncio. Asyncio [6] is a module that allows concurrent programming, though it is not concurrent. As explained in the background section, using the *await/async* method, a user can make a function wait that would otherwise be blocking and allow other function to run during the waiting period. The main issue with coroutines is that, when a coroutine *yield*s, the profilers receive a *return* event just like we exit from the function. That means that the time spent in the coroutine does not get accumulated and the call count get incremented since it is a regular function exit. This is a problem when measuring the wall time because the time in the coroutines does not get added. Yappi differentiates the *yield* from a real function exit and will accumulate the time correctly and correct the call count. A more detailed description of the problem is described in this note [7]. Hence, we chose Yappi as our time profiler. The module pstats [4] was used to format the profiling statistics into reports and snakeviz was used to visualize them.

### Tracemalloc & memory\_profiler

For the memory profiling, both profilers were used to get a better understanding of the performance. Memory\_profiler is a sampling profiler that uses the psutil library under the hood to inspect memory usage of the current process. Tracemalloc gives us a plethora of option to see how memory is allocated.

# Results

This section presents the results of the simulation framework in three stages. First, we show the runtime of the initial simulation and how it scales with more agents. Second, we analyse the results of the time and memory profiler and assess the bottlenecks in the framework. Finally, we outline the possible optimization that could be used to improve runtime and scalability using the Big O notation.

## Simulation results

The runtime of a simulation described in the methodology using nine residential agents and one agent0 is about 200 s. The simulation was run from 2 up to 100 agents to see how the simulation scales and it gave us the following result. There was no need to go beyond 100 agents to conclude as to how the simulation scaled, and it would be time wasted waiting for the simulation to finish.

*Figure 4: Graph representation of the simulation runtime in function of the number of agents*

The equation is for the trendline is:

With x the number of agents and y the simulation runtime.

That means that for every added agent, the runtime increases with 3.84 seconds. If this trendline continues, we would get the following time for the simulation of ten thousand agents.

This is about 124 times slower than our goal of a simulation runtime of 5 minutes.

We can conclude from the graph that the simulation scales linearly with a base runtime of about 120 seconds. This means that the system scales in a stable way and is not experiencing slowdowns but that has yet to be tested by running the simulation with more agents. It is also clear to see that, though the system is not experiencing slowdown, it is not accelerating either. As calculated above, simulating ten thousand agents would take a lot some time so optimization is necessary. We analysed which functions took up the most time and discussed how we can optimize them in the following paragraphs.

## Profiling

What is common is both time and memory profilers is that they produce significant amount of overhead. The simulation runtime doubles when using Yappi and almost quadruples when using tracemalloc. They were never used simultaneously when profiling, either only the time performance was profiled or only the memory consumption. The time performance was profiled more rigorously and in greater detail since it served as a greater bottleneck than the memory consumption as mentioned above. It will also be made clear in the description of the figures whether we profiled WALL time or CPU time.

### Afbeelding met schermopname, tekst, lijn Door AI gegenereerde inhoud is mogelijk onjuist.Time

*Figure 5: Snakeviz visualization of the time per function (WALL time)*

**Explain the time it took and what the core bs is .**The figure below shows the zoomed in figure of the interesting part, indicated by yellow box.

Afbeelding met tekst, schermopname, Kleurrijkheid, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist. *Figure 6: Detailed breakdown of the actual time profile*

The framework logic that is beyond the scope of the assume framework is not for optimization **modify**. The visualization works by denoting the function and amount of cumulative runtime it takes, meaning the time in the subcalls are also added. The subcalls/functions are the boxes below the function constrained by the width of the box. The following flow chart shows the path to the two main time-consuming functions that can potentially be optimized. This shows all the function below the orange box, but the names are not shown in the snakeviz visualization. Afbeelding met tekst, schermopname, Lettertype, diagram

Door AI gegenereerde inhoud is mogelijk onjuist.

*Figure 7: flow chart of the two main time-consuming functions*

The reason for leaving out the other functions that consume time like for example the other subfunctions of the *UnitsOperator.get\_actual\_dispatch*, is the following

As can be seen on figure 6, the other subfunctions all consists of multiple subfunctions, and this trend continues until a leaf function is reached. There is no gain with the time constraints to look for and implement an optimization for the leaf functions, the time would better be spent on the bigger time consuming functions. Additionally, ASSUME uses the Pandas library [https://pandas.pydata.org/docs/] to work with tabular data like SQL tables. Pandas is already high-performance but an ASSUME contains an implementation of the pandas library suited to the needs of the framework making it even more efficient. **more concise why the other functions not chosen**

The two main optimizable functions then are:

1. *aggregate\_step\_amount* in the utils.py script
2. *calculate\_cashflow* in the base.py script, BaseUnit class (Demand inherits from BaseUnit)

We then look at the detailed time performance for each of the functions and get the following figure.

*Figure 8: Top two optimizable functions and their detailed time performance*

The *ncalls* is the number of times the function is called, the *tottime* is the amount of time spent in the function excluding subfunctions and the *cumtime* is the amount of time spend in the function including subfunctions. Function (1) takes 381 microseconds to run and function (1) takes 37 microseconds to run. This tells us that the latter function is already very optimized. We therefore focus all out attention on the first function.

We focus on function (1) where a theoretical analysis using the Big O notation will be done because it is an extensive function and provide the optimizations. For function (2) a simple optimization will be tried and the profiled after to be compared in terms of performance.

#### Big O aggregate\_step\_amount

A time complexity analysis will be done of the aggregate\_step\_amount function. The function is divided into four sections where each section is denoted in figure 9 by the color that the section is highlighted with. The unhighlighted parts are variable instantiations.

1. Construction of deltas list
2. Sorting deltas
3. Construction of aggregation dictionary
4. Flattening of aggregation and return

The combined time complexity comes down to

But since the Big O only cares about growth rate, the dominant term is only one that matter.

### Memory

The figure below shows the top 5 memory allocation from a simulation with 10 agents.

* Yet to make clear

## Optimization

There were two optimizations for the aggregate\_step\_amount function and one optimization for the calculate\_cashflow function.

### Aggregate\_step\_amount function

The time complexity analysis showed that the major time-consuming component of the function was sorting. The default algorithm uses the built-in *sorted()* function that takes in any iterable and a potential *lambda* key and sorts it. The function does this by making a copy of the iterable and sorting the copy, leaving the original unchanged. This would be useful if we needed the original, but we do not. Various sources [x] compared the functions *list.sort()* and *sorted()*, and concluded that the first is 13% faster and consumes around 24% less memory. The only downside is that it can be only used on lists.

We then implemented this strategy in the code and got the following results.

Afbeelding met tekst, schermopname, Kleurrijkheid, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist.*figure x: time profiling after list.sort optimization*

A speed increase of about 13% compared to figure 6 was expected but instead we get about a 16% decrease in speed, an increase in time consumption. A potential reason for this outcome could be that the both sort function are based on Python Timsort that has a time complexity of O(n) for nearly sorted lists and that the 13% increase is rather for unsorted lists. We tested this hypothesis by shuffling the deltas before being sorted and got small improvement.

Doing the same calculation here, we get a simulation time for 10000 agents of 10 h 31 minutes which is an 11 minute time decrease.

After a thorough analysis of the structure and behavior of the function, we concluded that its complexity is inherent to the requirement it fulfills. The reasoning is as follows:

The function processes an order book of bids, each with a predefined start time, end time, and accepted volume. It constructs a timeseries representing the total active accepted volume at any given point within a specified timeframe. This aggregation is further disaggregated by grouping criteria (market ID and unit ID), yielding separate volume timeseries for each group. The result is a step function where volume levels change instantaneously as bids become active or expire. It is important to note that this function does not perform the market dispatch (i.e., it does not select which bids are accepted or determine clearing prices); rather, it aggregates the volumetric outcomes of bids whose acceptance and volumes are already determined.

The outcome is returned as a matrix with the time, aggregated volume and group. In the case of our scenario with no complex orders, the returned value just becomes the first bid in the order book in a different format. To speed up the code, we could remove the unnecessary creation of the timeseries *deltas*, the sorting and the cumulation of volumes but this would change structure of the function in for more complex orders. This is the reason that we concluded that an optimization for the *aggregate\_step\_amount* function in the case of our scenario is not attainable.

### Calculate\_cashflow function

Afbeelding met tekst, schermopname, Kleurrijkheid, Rechthoek

Door AI gegenereerde inhoud is mogelijk onjuist.The optimization here was to reduce frequent, small, indexed writes to a complex dictionary by accumulating locally and performing a single update. This, as show in figure x, did not decrease the runtime of the simulation. The reason for this can be that pandas data structure used for the local update adds overhead or that the function is not a major bottleneck in term of time consumption but rather in terms of call count.

# Conclusion

# Acknowledgements

# Appendices

Appendix 1

|  |
| --- |
| Function AGGREGATE\_STEP\_AMOUNT(orderbook, begin = None, end = None, groupby = None):  If groupby is None:  groupby ← empty list  deltas ← empty list  For each bid in orderbook:  group\_key ← tuple of values for each field in groupby  If bid has a single accepted\_volume:  Append (start\_time, +accepted\_volume, group\_key) to deltas  Append (end\_time, −accepted\_volume, group\_key) to deltas  Else if bid has a dictionary of accepted\_volumes:  duration ← (start\_time − end\_time) / number of accepted\_volume entries  For each (time, volume) in accepted\_volume:  Append (time, +volume, group\_key) to deltas  Append (time + duration, −volume, group\_key) to deltas  Else if bid has "only\_hours" defined:  Compute duration from only\_hours  Generate time slots between start\_time and end\_time at the specified hour range  For each generated time slot:  Append (start, +volume, group\_key) to deltas  Append (end, −volume, group\_key) to deltas  current\_power ← mapping of group\_key to 0  aggregation ← mapping of group\_key to empty list  Sort deltas by timestamp  For each (time, delta, group\_key) in deltas:  Update current\_power[group\_key] ← current\_power + delta  If time ∈ [begin, end]:  If last recorded time for group\_key == current time:  Overwrite last entry's power  Else:  Append (time, current\_power, group\_key) to aggregation  Flatten and return all entries in aggregation |

*Figure 9: the aggregate\_step\_amount algorithm*

# References

[1] Gross electricity generation per source 2022:

<https://www.iea.org/countries/belgium/electricity#where-does-belgium-get-its-electricity>

<https://en.wikipedia.org/wiki/Electricity_sector_in_Belgium>

[2] Smart meter profiles: <https://opendata.fluvius.be/explore/dataset/1_50-verbruiksprofielen-dm-elek-kwartierwaarden-voor-een-volledig-jaar/information/>

[3] Total load by all grid users: <https://opendata.elia.be/explore/dataset/ods001/table/>

[4] Python profilers: <https://docs.python.org/3/library/profile.html>

[5] Yappi: <https://pypi.org/project/yappi/>

[7] Coroutine problem with cProfile: <https://github.com/sumerc/yappi/blob/master/doc/coroutine-profiling.md>

[8] snakeviz: <https://jiffyclub.github.io/snakeviz/>

[1] Electricity market

<https://www.epexspot.com/en/basicspowermarket>

<https://www.next-kraftwerke.com/knowledge/day-ahead-trading-electricity>

<https://www.next-kraftwerke.com/knowledge/intraday-trading>

[2] ASSUME framework, Mango & Asyncio

<https://assume-project.de/#research>

Harder, N., Qussous, R., & Weidlich, A. (2023). Fit for purpose: Modeling wholesale electricity markets realistically with multi-agent deep reinforcement learning. Energy and AI, 14, 100295. doi: 10.1016/j.egyai.2023.100295

<https://realpython.com/async-io-python/>

[3] Big O<https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/>

Profiling

<https://www.ranorex.com/blog/what-is-code-profiling-and-how-to-choose-the-right-tool/>

<https://www.machinelearningplus.com/python/cprofile-how-to-profile-your-python-code/>

[x] list.sort and sorted

<https://www.geeksforgeeks.org/python-difference-between-sorted-and-sort/>

<https://medium.com/%40DahlitzF/list-sort-vs-sorted-list-aab92c00e17>

<https://stackoverflow.com/questions/1436962/python-sort-method-on-list-vs-builtin-sorted-function>