**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

Table of Contents

[1 Abstract 3](#_Toc198675226)

[2 Introduction 3](#_Toc198675227)

[3 Background 3](#_Toc198675228)

[3.1 The electricity market 3](#_Toc198675229)

[3.2 The ASSUME Framework 4](#_Toc198675230)

[3.3 Big O 4](#_Toc198675231)

[3.3.1 Math 4](#_Toc198675232)

[4 Methodology 4](#_Toc198675233)

[4.1 Initial simulation 4](#_Toc198675234)

[4.2 Profiling 5](#_Toc198675235)

[4.2.1 cProfile & yappi 5](#_Toc198675236)

[4.2.2 Tracemalloc & memory\_profiler 5](#_Toc198675237)

[5 Results 6](#_Toc198675238)

[5.1 Simulation results 6](#_Toc198675239)

[5.2 Profiling 6](#_Toc198675240)

[5.2.1 Time 7](#_Toc198675241)

[5.3 Optimization 8](#_Toc198675242)

[5.3.1 *Aggregate\_step\_amount* function 8](#_Toc198675243)

[*5.3.2* *Calculate\_cashflow* function 9](#_Toc198675244)

[6 Conclusion 9](#_Toc198675245)

[7 Acknowledgement 10](#_Toc198675246)

[8 References 10](#_Toc198675247)

# Abstract

**This research project explores the scalability and performance of the ASSUME framework for simulating a year of market operation with ten thousand residential agents within five minutes. The simulation approximates a model of the Belgian day-ahead electricity market using real-world production and consumption data. Initial profiling identified time as the greater bottleneck rather than memory consumption. Using deterministic profiling and visualization tools, we were able to identify the bottlenecks. Optimization strategies were proposed but did not result in a significant improvement. The study demonstrates that while ASSUME scales linearly for time, Significant optimization is necessary for large-scale, time-bound simulations and highlights future improvements toward realistic flex aggregator implementations.**

# Introduction

The goal of this project is to simulate a full year of electricity market operations involving ten thousand residential agents within five minutes on a model of the Belgian electricity market using the ASSUME framework. To achieve this, we will first create and simulate an approximate market scenario, profile its performance, and then optimize the framework to reduce simulation time. The research questions:

* What is the processing time and memory usage of the initial scenario?
* What are the main performance and memory bottlenecks?
* How can we optimize the ASSUME framework to simulate 10,000 agents in five minutes?

This research explores a future where residential consumers buy electricity directly from the market through a flex aggregator which is a service provider that pools and manages household flexibility (like shifting demand or using battery storage) to optimize costs and balance the grid. The goal is to simulate how this setup would work in practice and assess its performance before a real-world implementation can be realized. Flex aggregators enable better integration of renewable energy, reduce peak demand, and increase energy efficiency, making the electricity system more resilient and sustainable.

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 that 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 approximates 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

Unlike the stock market, the electricity market deals in non-storable, real-life commodity: electricity. Participants of the electricity market, producers and consumers, submit buy and sell orders to an orderbook. Based on this, a Market Clearing Price (MCP) is calculated using the merit order principle, dispatching the cheapest generation units first. There are two main markets:

* Day-Ahead Market: Operates via blind auctions once per day, trading hourly blocks for the next day.
* Intraday Market: Enables continuous trading up to 5 minutes before delivery in hourly, half-hourly, or quarter-hourly contracts.

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. 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 can be roughly divided into two parts. The markets are located on the left side of the world class and the market participant, which are named units, on the right side. Both are connected via the orders that market participants place on the markets.

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. [2]

## 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. [3]

### Math

#### Definition

# Methodology

An ASUS VivoBook laptop with a AMD Ryzen 7 4700U 8 core 2 GHz cpu and 16 GB of RAM was used for the work in this paper. The work is split into three sequential steps: initial simulation, profiling and optimization. It is important to note that the results of the performance analysis and the optimization was done using 10 agents, nine residential and the agent0. The reasoning is that ten agents show the bottlenecks well enough to not lose time waiting for a longer simulation to finish. This also means that the results that we obtain for simulating ten thousands agent are by means of extrapolation and could be somewhat inaccurate.

## 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 five 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 [4]. 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 1: 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 [5]. 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 [6]. 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 market is defined as a day ahead market for the year 2022 because this is the most recent year for which the Fluvius meter data is available.

## Profiling

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 [7] and Yappi [8] (Yet Another Python profiler).

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. Wall time is the actual real-world time a program takes to run (from start to finish). CPU time is the total time the CPU spends actively processing the program (excluding time spent waiting or idle). The ASSUME framework is built on top of the Mango framework which in turn is built on top of asyncio. Asyncio is a module that allows concurrent programming. 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 [9]. Hence, we chose Yappi as our time profiler. The module pstats was used to format the profiling statistics into reports and snakeviz [10] was used to visualize them.

### Tracemalloc & memory\_profiler

For the memory profiling, tracemalloc was 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. In contrast, tracemalloc provides a more detailed and advanced view by tracking memory allocations at the line level and showing the exact source of memory usage. We used it to examine peak memory allocation during runtime, helping us identify which parts of the code consumed the most memory.

# 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. Secondly, we analyse the results of the time and memory profiling and assess the bottlenecks in the framework. Finally, we outline the possible optimizations that could be used to improve runtime and scalability using the Big O notation.

## Simulation results

*Figure 2: Graph representation of (a) the simulation runtime and (b) the peak memory consumption in function of the number of agents*

The runtime T and peak memory usage M of the simulation using N = 10 agents is 197 s and 1.04 GB respectively. We let the number of agents N range from 5 to 100 (200 for T) and get a trendline with the equation: for time and for the memory usage. That means that for every added agent, the runtime increases with 3.84 seconds. Extrapolating this to N = 10000 agents, we would get a time of 10 h 42 min. and a memory usage of 5.3 GB.We can conclude from the graph (a) that the simulation scales linearly but too steep such that simulating ten thousand agents would take a considerate amount of time, making optimization necessary. The memory, though still large, is still feasible for the regular computer of this time so we decide not to look further into it. We analysed which functions took up the most time and discuss 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. Only wall time was simulated because the profile of the cpu time gave us no insight into the time bottlenecks of the code.

Visualization was done by the snakeviz library. The visualization shows an inverse stack where the top box takes up the most cumulative time and the boxes below it constrained with its width are its subfunctions that consume part of the time of the parent box. This structure is followed until a leaf box (box with no subfunctions) is reached. That means that for optimization, we should mostly look at the leaf boxes.

### Time

Afbeelding met schermopname, tekst, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.*Figure 3: Snakeviz visualization of the time per function*

Afbeelding met tekst, schermopname, Kleurrijkheid, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist.Figure 4 shows the zoomed in figure of the interesting part, indicated by yellow box in figure 3. The other boxes in figure 3 are part of the Mango framework that enable the agents communication. Most of the time is spent waiting for a message from the agent (bid or offer) which is received in the *check\_inbox()* method. The inbox itself is a *Queue,* hence why the function method is called. The runtime was about 400 s and this is multiplied by the number of agents (10) which is about the time shows in the purple and blue boxes. This is however not an indication of the total consumed time but rather a flawed visualization by snakeviz.

*Figure 4: Detailed breakdown of the actual time profile*

Afbeelding met tekst, schermopname, Lettertype, nummer

Door AI gegenereerde inhoud is mogelijk onjuist.The framework logic that is beyond the scope of the assume framework is not for optimization because it is minimal and straightforward, with no complex or resource-heavy operations that would benefit from optimization. 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.

*Figure 6: flow chart of the two main time-consuming functions*

We did not target certain functions for optimization (e.g. the other functions below *handle\_market\_feedback)* because they are mostly subfunction that call their own subfunctions until very small leaf function is reached. The cost of optimizing these trivial leaf function outweighs the performance gain. The other functions are calls to the frameworks optimized implementation of the Pandas library called *fast\_pandas,* so they require no further optimization either. The two main optimizable functions then are: *aggregate\_step\_amount* (1)in the utils.py script and *calculate\_cashflow* (2)in the base.py script, BaseUnit class (Demand inherits from BaseUnit)

We then look at the detailed time performance of two functions that we selected on figure 7.

*Figure 7: 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; this is the same as the cpu time. The *cumtime* is the amount of time spend in the function including subfunctions; this is the wall time. Function (1) takes 381 microseconds to run and function (1) takes 37 microseconds to run. We focus on function (1) where a theoretical analysis of the time complexity 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 profiled after to be compared in terms of performance.

#### Theoretical analysis of the time complexity of the *aggregate\_step\_amount* function

The function is divided into four sections where each section is denoted in figure 9 by The rectangles are variable instantiations.

Afbeelding met tekst, schermopname, Lettertype, ontvangst

Door AI gegenereerde inhoud is mogelijk onjuist.

Python built-in sorted function uses Timsort

**)**

*Figure 8: the aggregate\_step\_amount algorithm*

The total time complexity is , but since only the growth matter in the Big O notation, the constant and smallers terms are dropped making the time complexity T of the function *aggregate\_step\_amount:*

## Optimization

### *Aggregate\_step\_amount* function

The time complexity analysis revealed that the major time-consuming operation in the function is the sorting of the *deltas* list. By default, the function uses Python’s built-in *sorted()* function, which accepts any iterable and an optional key function. Internally, *sorted()* creates a copy of the input iterable and performs an in-place sort on that copy, leaving the original unchanged. Based on performance comparisons from various sources [11], we found that *list.sort(),* which sorts the list in-place, is approximately 13% faster than *sorted(),* though it can only be used on actual lists.

Therefore, we replaced *sorted()* with *list.sort()* and also moved filtering operations from the *deltas* loop to the earlier *bids* loop, which reduces the number of *delta* entries generated. Additionally, we replaced string-based dictionary keys with tuple keys, which are generally faster an. Despite implementing these strategies, the results did not meet expectations. Instead of an improvement, we observed an average 16% increase in execution time. A likely explanation is that both *sorted()* and *list.sort()* use Python’s Timsort algorithm, which has O(n) time complexity for nearly or fully sorted inputs. Since our *deltas* list is typically already close to sorted, the sorting operation is already highly optimized under Timsort. The 13% performance difference from the literature likely arises only in larger, unsorted lists, and does not translate to significant improvement in our use case.

To test this hypothesis, we shuffled the deltas list before sorting it, simulating a worst-case sorting scenario, and ran simulations with agents ranging from 5 to 200. Extrapolating from these results, the projected simulation time for 10,000 agents decreased by only 11 minutes (from 10h42m to 10h31m). This marginal gain could easily fall within the error margin of the estimation and does not represent a meaningful optimization.

### *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 9, 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.

*Figure 9: Profiling output after optimization of the calculate\_cashflow function*

# Conclusion

The results of the simulation and profiling demonstrates that while the simulation scales linearly in time, the slope of this scalability is too steep to handle large-scale scenarios efficiently. The peak memory usage scales logarithmically and while this does not scale as bad, it does still require a decent amount of memory. Simulating ten thousand agents would require over ten hours and six GB of memory, making it impractical without optimization. The profiling and optimization was rather focused on time performance due to it being a greater bottleneck.

Profiling revealed that the primary time bottlenecks lies in two computational functions: *aggregate\_step\_amount* and *calculated\_cashflow*. Despite targeted optimization like replacing *sorted()* with *list.sort()*, refactoring filter logic and optimizing dictionary updates for the *aggregate\_step\_amount* function, the improvement were marginal. Optimization for the *calculate\_cashflow* function was to reduce frequent, small, indexed writes to a complex dictionary by accumulating locally and performing a single update but this also yielded marginal results.

Ultimately, this work shows that while small-scale, code-level optimizations can offer useful insights, they do not significantly reduce runtime or solve the scalability challenges of the framework. A fully optimized solution was not reached. Future improvements should focus on higher-level architectural changes, such as asynchronous batching, parallel execution, or more efficient agent communication. These approaches hold more promise for improving performance and enabling large-scale simulation

# Acknowledgement

I would like to express my sincere gratitude to my promotor, Benoit De Vrieze and professor Hellinckx, for their continuous guidance and feedback throughout the course of this project.

# References

[1] *Basics of the Power Market | EPEX SPOT*. (2021). Epexspot.com. https://www.epexspot.com/en/basicspowermarket

[2] 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

Solomon, B. (2019, January 16). *Async IO in Python: A Complete Walkthrough*. Realpython.com; Real Python. https://realpython.com/async-io-python/

[3] GeeksforGeeks. (2018, February 28). *Big O Notation Tutorial A Guide to Big O Analysis*. GeeksforGeeks. https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

[4] Belgium - Countries & Regions - IEA. (2025). *Belgium - Countries & Regions - IEA*. IEA. https://www.iea.org/countries/belgium/electricity#where-does-belgium-get-its-electricity

to, C. (2011, March 24). *overview of the electricity sector in Belgium*. Wikipedia.org; Wikimedia Foundation, Inc. https://en.wikipedia.org/wiki/Electricity\_sector\_in\_Belgium

[5] *Verbruiksprofielen digitale elektriciteitsmeters: kwartierwaarden voor een volledig jaar.* (2022). Fluvius.be. https://opendata.fluvius.be/explore/dataset/1\_50-verbruiksprofielen-dm-elek-kwartierwaarden-voor-een-volledig-jaar/information/

‌[6] *Measured and forecasted total load on the Belgian grid (Historical data)*. (2024). Elia.be. https://opendata.elia.be/explore/dataset/ods001/table/?sort=datetime

[7] Python profilers: *The Python Profilers*. (2025). Python Documentation. https://docs.python.org/3/library/profile.html

[8] Yappi: *yappi*. (2024, November 12). PyPI. https://pypi.org/project/yappi/

[9]sumerc. (2025). *yappi/doc/coroutine-profiling.md at master · sumerc/yappi*. GitHub. https://github.com/sumerc/yappi/blob/master/doc/coroutine-profiling.md

‌[10] jiffyclub, M. D. (2025). *SnakeViz*. Github.io. https://jiffyclub.github.io/snakeviz/

[11]

GeeksforGeeks. (2019, December 31). *Python Difference between sorted() and sort()*. GeeksforGeeks. https://www.geeksforgeeks.org/python-difference-between-sorted-and-sort/

‌Florian Dahlitz. (2019, April 8). *list.sort() vs. sorted(list) - Florian Dahlitz - Medium*. Medium. https://medium.com/%40DahlitzF/list-sort-vs-sorted-list-aab92c00e17

‌method. (2009, September 17). *Stack Overflow*. Stack Overflow. https://stackoverflow.com/questions/1436962/python-sort-method-on-list-vs-builtin-sorted-function

‌[12] *ChatGPT*. (2025). Chatgpt.com. https://chatgpt.com/