

|  |  |
| --- | --- |
| **Faculty of Applied Engineering**  **Campus Groenenborger**  Groenenborgerlaan 171  2020 Antwerp | Role number: 20203834  Mobile phone: +32 489 69 80 37  Sameer.baruwal@uantwerpen.be |

|  |
| --- |
| **M4S-06: Software engineering for real-time electricity market bidding**  **Sameer Baruwal** |
|  |

**📘 1. Introduction**

**Goal**: Introduce your topic, justify its importance, and present your objectives.

**Include:**

* **Background**: Explain what the ASSUME framework is and its purpose in simulating energy markets.
* **Problem**: Describe the current limitations (e.g. scalability, performance bottlenecks with many agents).
* **Objective**: State clearly that you're simulating 10,000 agents and improving the framework's performance.
* **Research Question(s)**:
  + How can the ASSUME framework be optimized to simulate 10,000 agents efficiently?
  + What are the key performance bottlenecks?
* **Scope**: What is included (agent behavior, market simulation) and excluded (e.g., real-time deployment).
* **Structure**: Briefly describe what each chapter contains.

**📚 2. Literature Review**

**Goal**: Show what’s already known and how your work fits in.

**Include:**

* **Overview of Agent-Based Modeling (ABM)** in energy systems or markets.
* **Performance considerations** in large-scale simulations (CPU, memory, concurrency).
* **ASSUME framework**: Prior work using it, architecture, scalability.
* **Relevant technologies**:
  + Mango framework (if applicable)
  + Asyncio and concurrency in Python
  + Other ABM frameworks for comparison (e.g., MESA, GridLAB-D).
* **Gap**: Highlight the lack of large-scale performance studies or optimizations on ASSUME.

**🧪 3. Methodology**

**Goal**: Explain how you conducted your simulation and optimization.

**Include:**

* **Dataset**: Explain the 1,300 smart meter samples and how they’re used to model 10,000 agents.
* **Agent modeling**: How agents are initialized, behave, and interact with the market.
* **Simulation process**:
  + Describe the sequence of simulation steps (market interaction, order matching, etc.).
* **Performance analysis**:
  + Baseline profiling (e.g., what parts of the code are slow).
  + Tools used (e.g., cProfile, asyncio timing, logging).
* **Optimization strategies**:
  + Asynchronous task scheduling (e.g., batching I/O)
  + Data structure changes (e.g., NumPy vs. lists, pandas efficiency)
  + Parallelization, if any (e.g., multiprocessing or concurrent tasks)
* **Testing setup**: Machine specs, simulation parameters, and how performance was measured (e.g., run time, memory usage, scalability).

**📊 4. Results**

**Goal**: Present what you found — clearly and objectively.

**Include:**

* **Baseline results**: Performance metrics before optimization (runtime, memory usage, agent throughput).
* **Post-optimization results**: After applying improvements.
* **Comparisons**: Tables or graphs showing improvements (e.g., runtime vs. number of agents).
* **Scalability analysis**: How well the simulation handles increases in agent count.
* **Edge cases or failures**: If something didn’t scale well, mention it honestly.

**🧾 5. Conclusion**

**Goal**: Summarize findings, reflect on them, and suggest next steps.

**Include:**

* **Summary** of what you did and found.
* **Key insights**: What worked and what didn’t.
* **Answer to your research question**: Did you manage to efficiently simulate 10,000 agents? How?
* **Limitations**: Technical or methodological boundaries.
* **Future work**:
  + Improving inter-agent communication.
  + Optimizing data handling.
  + Distributed simulation support (e.g., cluster computing).
  + Real-time simulation and integration with real-world data.

# Abstract

This study’s main goal is to find the time and memory bottlenecks in the ASSUME framework and find a proper optimization for it.

Table of contents

[**Sameer Baruwal** 1](#_Toc197336312)

[Abstract 3](#_Toc197336313)

[Introduction 4](#_Toc197336314)

[Literature Review 5](#_Toc197336315)

[The electricity market 5](#_Toc197336316)

[The ASSUME framework 6](#_Toc197336317)

[Profiling & optimization 6](#_Toc197336318)

[Methodology 7](#_Toc197336319)

[Results 7](#_Toc197336320)

[Conclusion 12](#_Toc197336321)

[Acknowledgements 12](#_Toc197336322)

[References 12](#_Toc197336323)

# Introduction

**Use of pronouns, we ?**

This research delves into the ASSUME framework to simulate an energy market with 10 000 residential agents. The framework is an easy-to-use market simulation toolbox with integrated reinforcement learning methods, though these methods are not used.The framework was chosen so that we did not need to develop a market from scratch, which can be quite complex. The main goal of this project is to simulate 10 000 residential agents on the Belgian electricity market in 5 minutes. To do this, we need to optimize the ASSUME framework efficiently and decrease its simulation time. Therefore, we need to find the key performance bottlenecks by means of profiling. The research questions are:

* How can we optimize the ASSUME framework efficiently to simulate 10 000 agents?
* What are the key performance bottlenecks?

This research displays a futuristic scenario where residential consumers, instead of buying electricity from a company like Engie, can buy it directly from 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?**

The next section will discuss the literature relevant to this research project. It will discuss the working mechanism of the electricity market, how the ASSUME framework models that market and finally what optimization is and how it is (not) implemented into the framework. The section that follows discusses the ways I set up the project. This includes how I initially set up a simulation environment that models the Belgian market, how I analyzed the bottlenecks with various profiling techniques and the optimization methods that were considered to overcome the bottlenecks. The results section will be quite 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 and acknowledging the people that have helped us get to this point.

# 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 main electricity market in Belgium is the Epexspot. The members of the market 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 price is calculated which will be explained in a later paragraph. Once the trade is complete, the transaction is cleared and settled. Clearing ensures the proper fulfillment of each contract registered on the market. The clearing entity, which in the case of the Epexspot is the ECC, steps in and becomes the contractual partner for both the buyer and the seller to ensure the fulfillment of each trade and to mitigate counterparty risk. By matching supply and demand, the market ensures transparent and reliable prices and the market operators such as the ECC (via TSOs) make sure the electricity is delivered and paid.

The Epexspot operates in two markets, Day-Ahead and Intraday, that fulfill their own indispensable 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 orders are logged in by the members before the orderbook closes. The market established a demand curve based on the buy-orders and a supply curve based on the sell order, both for each hour of the following day. The market clearing price (MCP), which reflects the demand and supply, lies at the intersection of both curves. **Maybe a graph of the MCP?** The MCP, that is determined for each delivery period, applies to all buyers and sellers. All buyers who submitted volumes at a price higher than the MCP are executed for these volumes and pay the MCP, and all sellers who submitted volumes priced lower than the MCP are executed for these volumes and receive the MCP.

The Intraday market offers the possibility to trade even more in the sort 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.

The price formation process on the Epexspot Day-Ahead market follows 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 market clearing price. The least expensive units are usually renewables like wind and solar while the most expensive are gas and coal plants.

### The ASSUME framework

The ASSUME framework works with a World entity. Here, you can add multiple markets, producers and consumers to closely simulate a real-world example. In our case however, we work with a single day ahead market, 5 producers and 10000 consumers to approximate the Belgian electricity market.

A producer or consumer is defined by the Unit class with the following properties, by which they are differentiated: id, unit\_type, unit\_operator\_id, unit\_params and forecaster. The id is used to avoid duplicates and the unit\_type is used to differentiate whether it is a producer, ‘power\_plant’*,* or a consumer, ‘demand’. There are seven unit\_types defined, and the simulation will throw an error if you choose one that is not defined. The unit\_operator\_id acts as a RoleAgent from the mango framework where the units defined under this operator act as roles. Without going into much detail, the operator (RoleAgent) is activated and in turns activates all the units (roles). The unit\_params defines the market that the units belong to, the bidding strategy, the minimum or maximum power and the price at which they bid of offer.

**How to set up a simulation perhaps?**

### Profiling & optimization

Optimization is the process of modifying a software system to make some aspects of it work more efficiently or use fewer resources. In general, a program can be optimized so that it executes more quickly, uses less memory or even draws less power. Optimization often comes with trade-offs, where enhancing one metric may come at the expense of another. A very common one is the space-time trade-off where you make the trade between program runtime and memory usage (**consumption?)**. The goal is to find a balance with the resources that you have and the objective you are trying to reach. To find what exactly needs to be optimized, we must find the performance bottlenecks, the parts of the code that consume the most time and resources, by doing a performance analysis by ways of profiling. **Ways of optimizing like vectorization?**

Profiling is an analysis technique used to measure and analyze a program’s performance while it is running. 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.

**Does the Big O notation need to be explained?**

# Methodology

**Explain how a simulation is set up**

**Explain how and with what was profiled**

**Explain the different optimization techniques**

The Profiler that we used is Yappi (Yet Another Python Porfiler). This is a fast and unique profiler because it supports asyncio profiling

# Results

Results in termen van performance

**hoe groot is de impact van de lack of results aan het onderzoek**

**Results from initial simulation**

**Results from profiling**

**Result from optimization + comparison**

**General discussion**  
from sortedcontainers import SortedList  
  
def aggregate\_step\_amount(orderbook: Orderbook, begin=None, end=None, groupby=None):  
 *"""  
 Step function with bought volume, allows setting timeframe through begin and end, and group by columns in groupby.  
  
 Args:  
 orderbook (Orderbook): The orderbook.  
 begin (datetime, optional): The begin time. Defaults to None.  
 end (datetime, optional): The end time. Defaults to None.  
 groupby (list[str], optional): The columns to group by. Defaults to None.  
  
 Returns:  
 list[tuple[datetime, float, str, str]]: The aggregated orderbook timeseries.  
  
 Examples:  
 If called without groupby, this returns the aggregated orderbook timeseries  
 """* if groupby is None:  
 groupby = []  
 deltas = []  
  
 # SortedList, replace append with add and remove the sorted in the second loop  
 #deltas = SortedList([])  
  
 # first we are creating a list of tuples with the following form:  
 # start, delta\_volume, bid\_id, market\_id  
 for bid in orderbook:  
 add = ()  
 for field in groupby:  
 add += (bid[field],)  
 if bid["only\_hours"] is None and not isinstance(bid["accepted\_volume"], dict):  
 deltas.append((bid["start\_time"], bid["accepted\_volume"]) + add)  
 deltas.append((bid["end\_time"], -bid["accepted\_volume"]) + add)  
 elif isinstance(bid["accepted\_volume"], dict):  
 start\_hour = bid["start\_time"]  
 end\_hour = bid["end\_time"]  
 duration = (start\_hour - end\_hour) / len(bid["accepted\_volume"])  
 for key in bid["accepted\_volume"].keys():  
 deltas.append((key, bid["accepted\_volume"][key]) + add)  
 deltas.append((key + duration, -bid["accepted\_volume"][key]) + add)  
 else:  
 # only\_hours allows to have peak or off-peak bids  
 start\_hour, end\_hour = bid["only\_hours"]  
 duration\_hours = end\_hour - start\_hour  
 if duration\_hours <= 0:  
 duration\_hours += 24  
  
 starts = rr.rrule(  
 rr.DAILY,  
 dtstart=bid["start\_time"],  
 byhour=start\_hour,  
 until=bid["end\_time"],  
 )  
 for date in starts:  
 start = date  
 end = date + timedelta(hours=duration\_hours)  
 deltas.append((start, bid["volume"]) + add)  
 deltas.append((end, -bid["volume"]) + add)  
 aggregation = defaultdict(list)  
 # current\_power is separated by group  
 current\_power = defaultdict(lambda: 0)  
 for d\_tuple in sorted(deltas, key=lambda i: i[0]):  
 time, delta, \*groupdata = d\_tuple  
 groupdata\_str = "\_".join(groupdata)  
 current\_power[groupdata\_str] += delta  
 # we don't know what the power will be at "end" yet  
 # as a new order with this start point might be added  
 # afterwards - so the end is excluded here  
 # this also makes sure that each timestamp is only written  
 # once when iteratively calling this function  
 if (not begin or time >= begin) and (not end or time < end):  
 if aggregation[groupdata\_str] and aggregation[groupdata\_str][-1][0] == time:  
 aggregation[groupdata\_str][-1][1] = current\_power[groupdata\_str]  
 else:  
 d\_list = list(d\_tuple)  
 d\_list[1] = current\_power[groupdata\_str]  
 aggregation[groupdata\_str].append(d\_list)  
  
 return [j for sub in list(aggregation.values()) for j in sub]  
  
# Vectorization  
def aggregate\_step\_amount(orderbook: Orderbook, begin=None, end=None, groupby=None):  
 if groupby is None:  
 groupby = []  
  
 delta\_records = []  
  
 for bid in orderbook:  
 group\_values = tuple(bid[field] for field in groupby)  
  
 if bid["only\_hours"] is None and not isinstance(bid["accepted\_volume"], dict):  
 delta\_records.append((bid["start\_time"], bid["accepted\_volume"], \*group\_values))  
 delta\_records.append((bid["end\_time"], -bid["accepted\_volume"], \*group\_values))  
  
 elif isinstance(bid["accepted\_volume"], dict):  
 start\_hour = bid["start\_time"]  
 end\_hour = bid["end\_time"]  
 duration = (end\_hour - start\_hour) / len(bid["accepted\_volume"])  
 for ts, vol in bid["accepted\_volume"].items():  
 delta\_records.append((ts, vol, \*group\_values))  
 delta\_records.append((ts + duration, -vol, \*group\_values))  
  
 else:  
 start\_hour, end\_hour = bid["only\_hours"]  
 duration\_hours = end\_hour - start\_hour  
 if duration\_hours <= 0:  
 duration\_hours += 24  
 starts = rr.rrule(  
 rr.DAILY,  
 dtstart=bid["start\_time"],  
 byhour=start\_hour,  
 until=bid["end\_time"],  
 )  
 for date in starts:  
 start = date  
 end = date + timedelta(hours=duration\_hours)  
 delta\_records.append((start, bid["volume"], \*group\_values))  
 delta\_records.append((end, -bid["volume"], \*group\_values))  
  
 # Construct DataFrame  
 column\_names = ["time", "delta"] + groupby  
 df = pd.DataFrame(delta\_records, columns=column\_names)  
 # print("delta records: ", delta\_records)  
 # print("df: ", df)  
  
 # Create a group identifier string column  
 if groupby:  
 df["group"] = df[groupby].astype(str).agg("\_".join, axis=1)  
 else:  
 df["group"] = "all"  
  
 df = df.sort\_values("time")  
  
 # Compute cumulative power  
 df["current\_power"] = df.groupby("group")["delta"].cumsum()  
  
 # Filter by time range  
 if begin:  
 df = df[df["time"] >= begin]  
 if end:  
 df = df[df["time"] < end]  
  
 # Prepare output: list of tuples [time, current\_power, group\_fields...]  
 output\_cols = ["time", "current\_power"] + groupby  
 result = list(df[output\_cols].itertuples(index=False, name=None))  
 #print("result: ", result)  
  
 return result

Optimization techniques and their time savings

Test on a work computer with processor = intel i5-12400 6 cores, 12 logical processors, 8 gb ram

10 agents

Normal run = 120 s

Yappi = 320 s -> about 200 s of overhead?

Optimization 1: SortedList -> sort the deltas -> the built-in sorted is not necessary anymore

Normal run = 122 s, yappi = 355 s

Optimization 2: Vectorization -> convert deltas to pandas dataframe to use sort and cumsum

Normal run = 286 s, yappi = I did not let it finish

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:30: size=92.8 MiB, count=419328, average=232 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\utils.py:98: size=8190 KiB, count=209664, average=40 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\utils.py:97: size=8190 KiB, count=209664, average=40 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:34: size=4914 KiB, count=209664, average=24 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:33: size=4914 KiB, count=209664, average=24 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\numpy\\_core\numeric.py:352: size=3359 KiB, count=152, average=22.1 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\base\_market.py:707: size=1784 KiB, count=1, average=1784 KiB

<frozen importlib.\_bootstrap\_external>:753: size=1627 KiB, count=12887, average=129 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\fast\_pandas.py:201: size=1100 KiB, count=8762, average=129 B

C:\Users\samee\AppData\Local\Programs\Python\Python312\Lib\copy.py:143: size=1097 KiB, count=45, average=24.4 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\fast\_pandas.py:317: size=1097 KiB, count=39, average=28.1 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\sqlalchemy\engine\default.py:1484: size=669 KiB, count=5971, average=115 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\utils.py:99: size=547 KiB, count=8758, average=64 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\fast\_pandas.py:985: size=343 KiB, count=15, average=22.9 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\dateutil\rrule.py:886: size=341 KiB, count=8740, average=40 B

<frozen importlib.\_bootstrap>:488: size=286 KiB, count=2264, average=129 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\pandas\core\frame.py:12683: size=274 KiB, count=12, average=22.9 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\xarray\coding\times.py:508: size=274 KiB, count=8, average=34.3 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\base\_market.py:255: size=256 KiB, count=1, average=256 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:155: size=205 KiB, count=8759, average=24 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:29: size=205 KiB, count=8759, average=24 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:19: size=205 KiB, count=8759, average=24 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\markets\clearing\_algorithms\simple.py:18: size=205 KiB, count=8759, average=24 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\units\_operator.py:381: size=189 KiB, count=2000, average=97 B

<frozen abc>:123: size=139 KiB, count=1719, average=83 B

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\xarray\core\duck\_array\_ops.py:419: size=137 KiB, count=7, average=19.6 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\pandas\core\arrays\\_ranges.py:88: size=137 KiB, count=4, average=34.3 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\fast\_pandas.py:605: size=137 KiB, count=4, average=34.3 KiB

C:\Users\samee\PycharmProjects\assumption\.venv\Lib\site-packages\assume\common\fast\_pandas.py:442: size=119 KiB, count=867, average=141 B

# Conclusion

**What do I conclude?**

# Acknowledgements

**Love**

# References

**APA 7?**

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

Basics of the Power Market | EPEX SPOT. (2025, May 02). Retrieved from <https://www.epexspot.com/en/basicspowermarket>

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

(for the "agent 0")

Day ahead forecast of all generation per type: https://opendata.elia.be/explore/dataset/ods034/information/

Actual generation per type: https://opendata.elia.be/explore/dataset/ods033/information/

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

<https://www.epexspot.com/en/market-results>

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

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

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

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

<https://www.sciencedirect.com/science/article/pii/S2666546823000678?via%3Dihub>

[https://docs.python.org/3/library/timeit.html](https://eur01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fdocs.python.org%2F3%2Flibrary%2Ftimeit.html&data=05%7C02%7CSameer.Baruwal%40student.uantwerpen.be%7Cf03beb0371cb425ca27f08dd65541ebc%7C792e08fb2d544a8eaf72202548136ef6%7C0%7C0%7C638778134131775603%7CUnknown%7CTWFpbGZsb3d8eyJFbXB0eU1hcGkiOnRydWUsIlYiOiIwLjAuMDAwMCIsIlAiOiJXaW4zMiIsIkFOIjoiTWFpbCIsIldUIjoyfQ%3D%3D%7C0%7C%7C%7C&sdata=C57XPsl9osXPqRih2tfTR4uFLSSvXZWFGR0%2BweVvcLs%3D&reserved=0)

<https://docs.python.org/3/library/profile.html>

<https://docs.python.org/3/library/tracemalloc.html>

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

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

<https://pypi.org/project/sortedcontainers/>

**How extensive does references need to be? Are the references to stuff like sortedcontainers necessary**