GPO-D

Graph-based Prescriptive Optimization with Dataflows

Project Report

cs-25-dt-10-01

Aalborg University
Computer Science CS-IT10

Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
GPO-D: Graph-based Prescriptive Optimization with Dataflows

Theme:
Prescriptive analytics

Project Period:
Spring Semester 2025

Project Group:
cs-25-dt-10-01

Participant(s):
Bence Schoblocher
Samuel Ivan
Simeon Kolev

Supervisor(s):
Torben Bach Pedersen
Martin Moesmann

Page Numbers: 74

Date of Completion:
June 6, 2025

Abstract:

Prescriptive analytics (PSA) combines data processing, machine learning, and mathematical optimisation. However, the existing PSA tools do not always support all of the above-mentioned parts, may be more specialised towards the optimisation part only, or are limited in terms of user-friendliness or developer productivity. This thesis introduces GPO-D (Graph-based Prescriptive Optimisation with Dataflows), a Python library aimed at simplifying PSA workflows using graph-based optimisation modelling and dataflow processes. GPO-D integrates the full PSA workflow with the aim of improving developer productivity. It adopts the concept of Directed Acyclic Graphs from Apache Airflow for managing dataflows and uses CVXPY for solving optimisation problems. Later on, a microgrid problem example is presented, for scheduling solar production, battery charge and discharge cycles. The solution to this example is then used as a base for evaluating the performance and productivity metrics compared to other Python libraries like CVXPY, Pyomo and GBOML. The results show that it achieves similar performance while decreasing the amount of code required.

Summary

Data analytics is often described in three main parts - descriptive analytics, predictive analytics and prescriptive analytics (PSA). PSA, as the most advanced stage of data analytics, aims to produce some actionable steps in the present, in the form of a prescription. This prescription is based on predictions describing the future that come from predictive analytics. Predictions are then based on processed historical data, which describe the past and come from descriptive analytics. This workflow, referred to as the full PSA workflow in this thesis, involves data processing, machine learning (ML) and optimisation problem-solving steps.

There are many specialised tools in programming languages like Python or R that focus on doing one of the steps from the full PSA workflow very well, which can be data processing, ML, or mathematical optimisation problem (MOP) solving. However, there is few tools that can do all of it within one ecosystem like a software, tool or library. One can potentially implement a full PSA solution in Python using Pandas, TensorFlow and CVXPY, but this still requires some integration efforts before the solution is up and running. Furthermore, this kind of glued-together solution is not very extensible to other PSA problems.

This thesis introduces GPO-D (Graph-based Prescriptive Optimisation with Dataflows) which is a Python library developed with the idea of increasing developer productivity and user-friendliness through abstractions that can combine all the phases in PSA, thus supporting the full PSA workflow. This is achieved with the implementation of *GraphProblemClass* and *Dataflow*. The *GraphProblemClass* further simplifies the definition and construction of optimisation problems by segmenting them into smaller parts called *domain nodes*. These domain nodes can extend the base implementation of the *Node* class or inherit the characteristics from each other. The MOP construction that happens inside the *GraphProblemClass* is then translated into the Python optimisation library, CVXPY in this case, and solved.

The domain we are dealing with is the (electricity) energy sector. The underlying *domain node* classes are based around the Harmonized Electricity Market Role Model (HEMRM)

published by ENTSO-E, EFET and ebIX. These classes focus on easy extensibility and standardisation. Furthermore, the GPO-D library is presented through an optimisation problem example about microgrids, which introduces solar panels and wind turbines as producers, households (homes) and a school as consumers and batteries to store the unused energy. Weather forecasts and historical production data can be used to generate predicted production curves with PyCaret, which is an automated machine learning library. Later on, the same library is used to predict future consumption for households based on historical usage data. This data is then fed into the optimiser through the *domain node* classes to generate an optimal strategy for the microgrid example.

The core prerequisite when solving an MOP is to have accurate and relevant data. That is what forms the basis for all decision-making. To be able to obtain such data, we need another well-defined abstraction. That is how we come up with the so-called *Dataflow*. We adopted the concept of Directed Acyclic Graph (DAG) from Apache Airflow, which is what the *Dataflow* is based on. In GPO-D, two core classes are introduced to model these *dataflows* - *Dataflow* and *DataflowTask*. The *DataflowTask* (or just *task*) represents a single unit of work in the *dataflow*, whereas the *Dataflow* class manages a list of *tasks*. Each (domain) *Node* that is part of the *GraphProblemClass* can be assigned a *dataflow*. That is the main mechanism we use to define how each *node* will receive its relevant data. The execution of *dataflows* is controlled by a *DataflowManager*, which is a helper class to manage and coordinate all *dataflows* within the application.

The experiments are designed to assess the performance of GPO-D and its impact on developer productivity when implementing the microgrid problem example. Specifically, we compare GPO-D against other libraries such as GBOML, CVXPY, and Pyomo with respect to those two metrics. Furthermore, we evaluate the scalability of GPO-D by progressively increasing the size of the microgrid optimisation problem to determine its performance limits.

Acknowledgements

We would like to express our sincere gratitude to our main supervisor, Prof. Torben Bach Pedersen, for their expert guidance, continuous support, and valuable feedback throughout the course of this thesis.

We are also thankful to our co-supervisor, Martin Moesmann, for their insightful suggestions and constructive comments during the development of this work.

We would especially like to acknowledge Prof. Wolfgang Lehner, whose participation in all supervisory meetings and ongoing input greatly contributed to the depth and direction of our research, despite not being formally listed as a supervisor.

Contents

Chapter 1

Introduction

In today's data-intensive world, the ability to transform large amounts of information into actionable decisions is crucial for competitive advantage. Descriptive and predictive analytics have become widely adopted across many industries and organisations, but their primary use is for explaining past events and forecasting future trends. This traditional decision-making approach of relying on human intuition and static reports is proving to be insufficient in this rapidly developing environment [1]. For that reason, Prescriptive analytics (PSA) comes into play, as it aims to recommend or even automate the decision-making, based on available data, constraints, and desired outcomes [2]. It represents the latest and most advanced stage of analytics, combining both optimisation, machine learning, and real-time data processing to assist human decision-making [2, 3, 1].

Despite the popularity and potential of PSA, its adoption in analytics systems remains scarce. A growing body of research has identified gaps in their integration, usability, scalability, and adaptability to real-world problems [2, 1]. This raises the need for tools that are accessible to users and domain-flexible, so that they can align with organisational workflows [3]. This paper contributes to this growing field by presenting an extensible graph-based solution for creating PSA workflows, highlighting how modern tools and libraries can be orchestrated to turn raw data into optimised decisions. Hereby, we present GPO-D (Graph-based Prescriptive Optimisation with Dataflows) (pronounced "gee-pod"), a Python-based library for PSA applications.

As a running example, we consider a renewable energy optimisation problem. We have a singular household with a PV (Photovoltaic) installation and a battery. For simplicity, properties such as battery capacity, battery charge/discharge power, battery efficiency, and PV production capacity are left out. Furthermore, the household is connected to the electrical grid, where produced (from the PV) and consumed (from the grid) electricity is measured via a metering point. Figure 1.1 depicts the example. The goal in this optimisation problem is to minimise the usage of the electrical grid, or conversely, maximise the

usage of the PV installation and the battery. Let's assume the scenario where the weather is sunny for most of the day until 18 PM, but then becomes cloudy. Table 1.1 shows various periods of the day, and the respective load, production, and battery state of charge at each period. For demonstration purposes, we assume that the battery starts off empty. Early in the morning, the solar panel will not be able to produce enough energy to satisfy the household's load. That forces the household to draw electricity from the electrical grid in order to meet the load (as seen in Table 1.1 from 6 to 9 AM). From 18 PM, again the solar panel is not able to produce enough energy to meet the load due to changing weather conditions. But since the battery has been charged during the day due to overproduction (higher production than load), the household will use energy from the battery instead of the electrical grid. The actual values for the household load and the solar production for tomorrow cannot be known, and the only other action is to try to predict them as accurately as possible. Therefore, the solution to this problem requires a complex workflow that combines data processing, machine learning, and optimisation, in order to find the most optimal energy usage for the solar production, battery, and the grid.

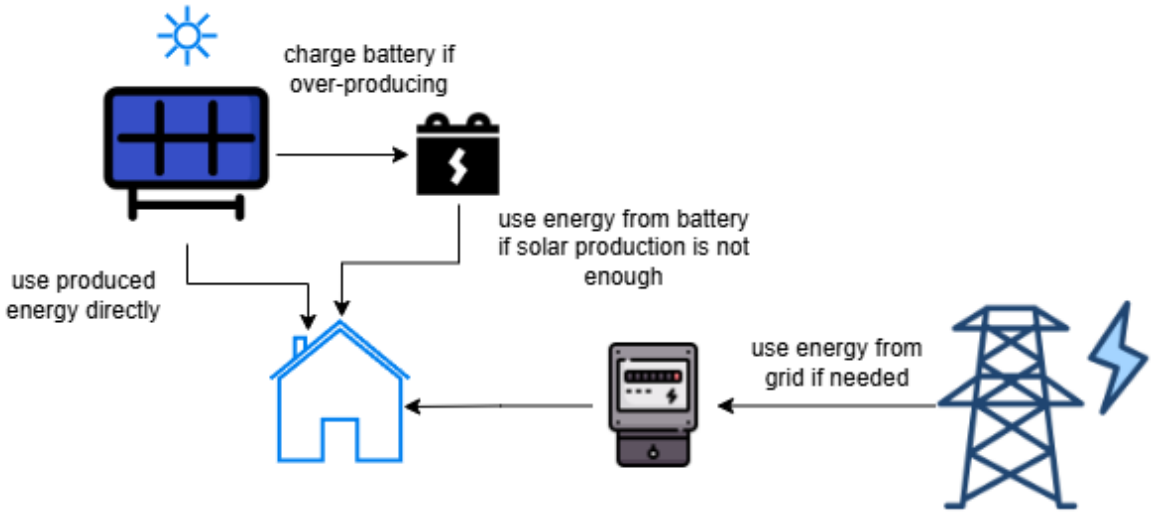

Figure 1.1: Household with a PV installation and a battery example. Icons made by Freepik and Iconjam from Flaticon.com [4, 5]

Timestamp	Load (kWh)	PV Production (kWh)	Battery SoC (kWh)	Grid Usage (kWh)
2025/05/28 06:00	0.8	0.0	0.0	0.8
2025/05/28 09:00	1.2	2.5	1.3	0.0
2025/05/28 12:00	1.0	3.8	4.1	0.0
2025/05/28 18:00	2.5	0.5	2.1	0.0
2025/05/28 21:00	1.8	0.0	0.3	1.5
2025/05/29 00:00	?	?	?	?

Table 1.1: Energy Usage and Production Data

We now show how this problem can be solved using GPO-D. Listing 1.1 shows a minimal working example, illustrating the overall structure and workflow of the solution. First, we define all the components of the problem (lines 1–5). Next, we specify how the data processing and forecasting should occur (lines 7–10). Finally, these elements are assembled into a problem instance, which is then solved by invoking the *solve* method of the problem instance (lines 12–18). This example shows how GPO-D abstracts away low-level solvers and domain-specific details, simplifying the integration of data processing, machine learning, and optimisation into a unified, solvable workflow.

```
1  problemClass = GraphProblemClass("running_example", time_length=24)
2  household = Consumer(...)
3  solar_panel = SolarPanel(...)
4  battery = Battery(...)
5  metering_point = MeteringPoint(...)
6
7  household.dataflow.task("predict_consumption", ...)
8  task1 = solar_panel.dataflow.task("get_irradiation_data", ...)
9  task2 = solar_panel.dataflow.task("generate_solar_data", ...)
10 task1 >> task2
11
12 problemClass.add_nodes([household, battery, solar_panel,
13                           metering_point])
14
15 metering_point.connect_to([household, solar_panel, battery])
16
17 result = problemClass.solve(solver=cp.CBC, objective="minimize",
18                              value="cost")
```

Listing 1.1: Running example solved with GPO-D

In this paper, we describe the process of designing and developing GPO-D for the above

and similar problems. In Chapter 2, we introduce similar papers and libraries related to GPO-D. In Chapter 3, we introduce the prescriptive analytics workflow and the software requirements for GPO-D. In Chapter 4, we state the goals and set a scope for this thesis. In Chapter 5, we introduce a detailed example that builds upon the running example, which we solve and implement with GPO-D in the next chapters. In Chapter 6, we describe the library design process as well as implementation choices. In Chapter 7, we compare the usability of GPO-D against other solutions, and we measure its scalability to test the limits. Finally, we conclude our findings in Chapter 8, as well as discuss possible future work.

1.1 Declaration

Generative AI tools were used to help with summarisation of papers and note taking, specifically ChatGPT, however, these references were cross-checked with the original material to ensure correctness and proper attribution of credit. Furthermore, GitHub Copilot was used to assist in the code implementation.

Chapter 2

Related Work

In this chapter, we will be looking at tools out of which we incorporate concepts and try to build upon them in this thesis. Specifically, the tools are:

- SolveDB and SolveDB+
- GBOML
- Apache Airflow

2.1 SolveDB and SolveDB+

SolveDB, presented by Šikšnys and Pedersen [6], is an RDBMS with integrated support for predictive (PDA) and prescriptive (PSA) analytics. The core idea with SolveDB is to unify data management and optimisation problem solving within a single tool, i.e. a DBMS in this particular case [6]. It does so through an SQL-based syntax with specific keywords for optimisation problems. More specifically, here are the new SQL keywords: [6]

- *SOLVESELECT* - formulation clause producing output relation from input relation with decision variables according to problem formulation in MINIMIZE/MAXIMIZE, SUBJECTO and USING clauses
- *MINIMIZE/MAXIMIZE* - specifies the objective function to minimize/maximize
- *SUBJECTTO* - specifies constraints applied to variables of an input relation
- *USING* - specifies the optimisation solver to be used for solving the objective function

```
1 SELECT * FROM (
2  SOLVESELECT battery_ch, battery_disch, soc, grid IN (
3   SELECT ts, load, pv_prod
4   FROM energy_data
```

```
5  ) as u
6  MINIMIZE (SELECT SUM(grid) FROM u)
7  SUBJECTTO
8      -- Load must be met by pv_prod + grid + battery_disch
9      (SELECT load <= pv_prod + grid + battery_disch FROM u),
10     -- Non-negative battery SoC
11     (SELECT 0 <= soc FROM u)
12     -- Non-negative battery charge/discharge
13     (SELECT 0 <= battery_ch FROM u)
14     (SELECT 0 <= battery_disch FROM u)
15     -- Non-negative grid usage
16     (SELECT 0 <= grid FROM u)
17 USING solverlp) AS s
```

Listing 2.1: Running example formulated in SolveDB.

Listing 2.1 shows an example solution to the running example problem done in SolveDB. The *SOLVESELECT* clause specifies *battery_ch*, *battery_disch*, *soc*, and *grid*, all of which represent decision variables together with input *SELECT * FROM energy_data* and output *as u* relations. Here we assume that a table *energy_data* exists and contains data for solar production and load of a household. The objective is to *MINIMIZE* the total grid usage, which is defined as *SUM(grid)* in the code. The constraints for the optimisation problem are specified under the *SUBJECTTO* clause as:

- the household's electricity demand must be met by the combination of electricity produced by the PV, electricity taken from the grid, and electricity discharged from the battery:
  (*SELECT load* $\leq$ *pv_prod + grid + battery_disch FROM u*)
- the battery's state of charge cannot go below 0 kWh. In this example, battery capacity is left out, therefore, no upper bound constraint is specified:
  (*SELECT 0* $\leq$ *soc FROM u)*
- the battery can only be charged or discharged in positive amounts (or not at all). Charging and discharging limits are left out in this example, therefore, no constraints are specified regarding that:
  (*SELECT 0* $\leq$ *battery_ch FROM u* and *SELECT 0* $\leq$ *battery_disch FROM u*)
- any electricity drawn from the grid must be zero or more (non-negative):
  (*SELECT 0* $\leq$ *grid FROM u*)

Lastly, *USING* keyword specifies which solver is to be used for solving our problem, which in this case is just the default *solverlp*. Behind the scenes, the actual black-box (in the paper

[6] labelled as *physical*) solver is automatically selected based on the suitability for the problem at hand.

Meanwhile, SolveDB+ [7] extends the original SolveDB solution with a predictive framework, shared optimisation models, and new language features, additionally improving the support for full PSA workflow (which includes data processing, machine learning and mathematical optimisation) as well as the overall performance. This makes SolveDB+ "the only tool as of 2021 that can unify prediction and optimisation problem solving within a SQL-based system." [7]

2.2 GBOML

GBOML, which stands for Graph-Based Modelling Optimisation Language, is a modelling language for mathematical programming implemented in Python and was first introduced in 2021 by Berger et al. paper [8]. It "enables the implementation of mixed-integer linear programs (MILP) found in different sectors such as energy or supply chain." [9] Generally, GBOML is used for implementing problems that involve dynamic mathematical optimisation over a finite time horizon [10]. Moreover, it is widely used for scenarios which can be described in a hypergraph-like structure of interconnected nodes, for example, microgrids optimisation, or transportation and logistics optimisation [9]. GBOML "uses features of both algebraic modelling language (AML) and object-oriented modelling language (OML)." [10]

An example solution to the running example is showcased in Listing 2.2 and Listing 2.3, which are done by following [9] and [10].

```
#TIMEHORIZON
T = 24;

#NODE DEMAND
#PARAMETERS
total_demand = import "../../data/demand.csv";
#VARIABLES
external: consumption[T];
#CONSTRAINTS
consumption[t] == total_demand[t];

#NODE SOLAR_PV
#PARAMETERS
max_power_output_kW = import "../../data/gen_solar.csv";
#VARIABLES
external: electricity[T];
```

```
#CONSTRAINTS
electricity[t] >= 0;
electricity[t] == c[t] * max_power_output_kW[t];

#NODE BATTERY
#PARAMETERS
SoC = 0;
#VARIABLES
internal: energy[T];
external: charge[T];
external: discharge[T];
#CONSTRAINTS
energy[t] >= 0;
charge[t] >= 0;
discharge[t] >= 0;
energy[t+1] == energy[t] + charge[t] - discharge[t];
energy[0] == SoC;

#NODE METERING_POINT
#VARIABLES
external: power_import[T];
#CONSTRAINTS
power_import[t] >= 0;
#OBJECTIVES
min: power_import[t];

#HYPEREDGE POWER_BALANCE
#CONSTRAINTS
SOLAR_PV.electricity[t] + BATTERY.discharge[t] +
METERINGPOINT.power_import[t] == BATTERY.charge[t] + DEMAND.consumption[t];
```

Listing 2.2: Running example formulated in GBOML

In Listing 2.2, the running example problem is modelled in the form of *nodes* (e.g. *SOLAR_PV*) and *hyperedges* (e.g. *POWER_BALANCE*) which represent the elements of the renewable energy optimisation problem. For a better overview, GBOML grammar contains the following keywords [10]:

- *#TIMEHORIZON* - length of optimization time horizon
- *#GLOBAL* - global parameters that can be accessed anywhere in the model file

- *#NODE* - represent optimisation subproblems with its set of parameters, variables (*internal* or *external*), constraints and objective function.
- *#HYPEREDGE* - can define its set of parameters, but it primarily couples variables belonging to multiple nodes through constraints

```
from gboml import GbomlGraph

gboml_model = GbomlGraph(24)
nodes, edges, _ = gboml_model.import_all_nodes_and_edges("running_example.txt")
gboml_model.add_nodes_in_model(*nodes)
gboml_model.add_hyperedges_in_model(*edges)
gboml_model.build_model()
solution = gboml_model.solve_cplex()
```

Listing 2.3: GBOML example execution in Python. The example is based on [10]

Secondly, in Listing 2.3, nodes and edges from the model are imported into Python and added into the *GbomlGraph* class. Then the model is built through *build_model()* method. Lastly, to the solution of the problem, a specific solver method is called, which in this example is CPLEX. At the moment, the package provides both commercial and open-source solvers, namely CPLEX (commercial), Cbc/Clp (open-source), DSP (experimental open-source), Gurobi (commercial), HiGHS (open-source), and Xpress (commercial). [10]

2.3 Apache Airflow

Apache Airflow is an open source, scalable, dynamic and extensible platform to schedule and monitor workflows [11]. By using Python alongside its user interface, Apache Airflow provides more flexibility, workflow management and ease of use for creating workflows and generating tasks for data-intensive systems [11]. Some of the use cases implemented with Apache Airflow can be business operations, Extract Transform Load (ETL)/Extract Load Transform (ELT), infrastructure management or Machine Learning Operations (MLOps) with ETL/ELT being the most common with over 85% use cases in 2024 based on the Airflow Survey [12]. The core idea of the platform is the Workflow as Code paradigm that is achieved through Directed Acyclic Graphs (DAGs) [13] as seen in figure 2.1.

```
1. with DAG(
2.     "example_dag",
3.     default_args={"retries": 2},
4.     schedule="@daily",
5.     start_date=pendulum.datetime(2023, 1, 1,tz="UTC")
6. ) as dag:
7.     def extract(**kwargs):
8.         # Python code logic goes here
9.     def transform(**kwargs):
10.         # Python code logic goes here
11.     def load(**kwargs):
12.         # Python code logic goes here
13.     extract_task = PythonOperator(
14.         task_id="extract",
15.         python_callable=extract,
16.     )
17.     transform_task = PythonOperator(
18.         task_id="transform",
19.         python_callable=transform,
20.     )
21.     load_task = PythonOperator(
22.         task_id="load",
23.         python_callable=load,
24.     )
25.     extract_task >> transform_task >> load_task
```

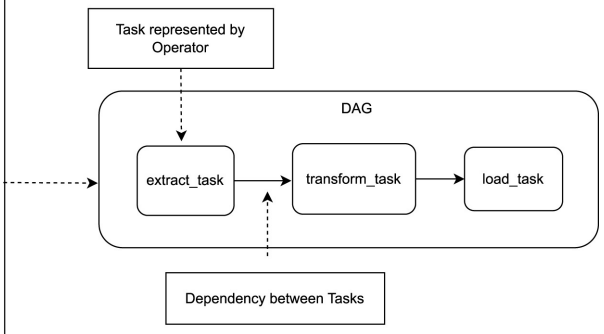

Figure 2.1: Apache Airflow DAG example taken from Yasmin et al. study [13]

The DAGs represent an order of execution for tasks associated with it where tasks themselves are nodes with directed edges representing dependencies [13]. Furthermore, different types of tasks are often implemented through "*operators* that encapsulate the necessary logic." [13]

For our running example, Figure 2.2 shows an example DAG for getting a forecast for the PV's production. In a similar fashion, we can design a DAG for getting a forecast of the household's load. Both DAGs can be implemented in code as shown in Figure 2.1.

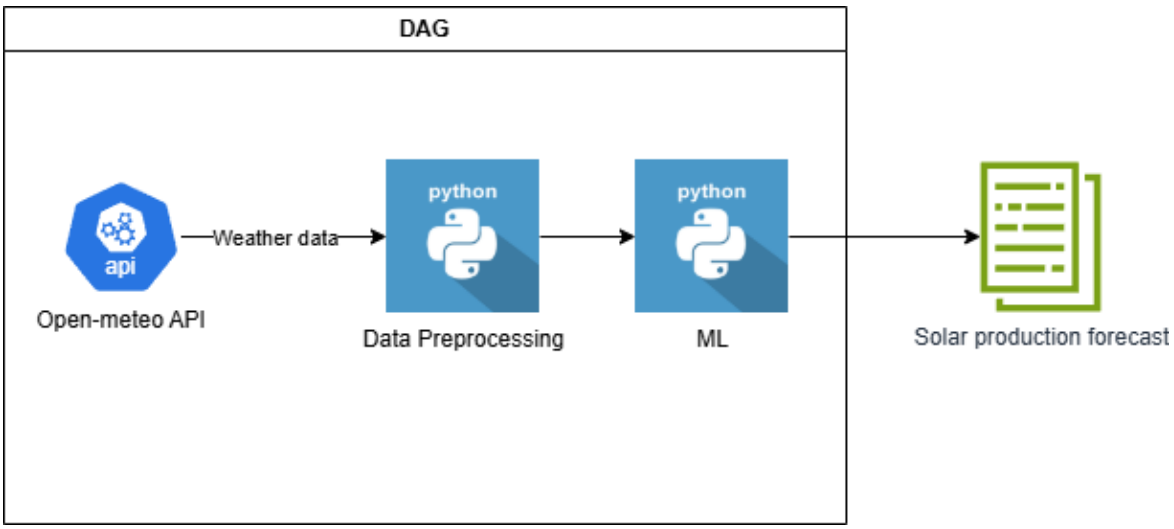

Figure 2.2: Apache Airflow DAG for running example

2.4 Summary

As we have seen in this chapter, there are tools such as SolveDB and SolveDB+ that consolidate data management and PSA workflows in one ecosystem. This idea of bringing them together also produced a positive impact on improving "usability, explainability, developer productivity, and performance, even for new users" [7]. That was confirmed in the usability study conducted by Laurynas Siksnys et al. and described in their SolveDB+ paper [7]. Secondly, GBOML, which focuses only on the mathematical optimisation part of the PSA workflow, introduces a concept of using an OML together with an AML. For this project, the object-oriented modelling concept in particular is the interesting part and one that we use some inspiration from. This idea makes it easier to reuse a "model" (where parameters, variables, constraints and objectives are defined) such as *BATTERY* or *SOLAR_PV* and reference it in other parts of a problem definition where needed, e.g. *POWER_BALANCE* as seen in Listing 2.2. Lastly, Apache Airflow uses the DAG model for creating ETL or MLOps workflows, which can make it easier and more flexible to abstract and unify the data processing with the machine learning and mathematical optimisation workflows in GPO-D.

Improvements over the listed technologies

- **SolveDB** and **SolveDB+**: These libraries are both in SQL and do not provide support for advanced machine learning tasks in Python.
- **GBOML**: Provides a mathematical optimisation using a graph-based model, but does not provide integrated support for data processing and machine learning.
- **Apache Airflow**: Schedules and manages workflows; it is not designed for prescriptive analytics.

We address these gaps by:

- Providing a unified Python interface for the whole PSA workflow.
- Using graph-based modelling, but in Python, making it easier to use in PSA workflows.
- Providing modular code that the user can extend or adapt to new domains.
- Providing dataflow management, which is inspired by Airflow.

Overall, in Section Research Gap 3.2 more detailed explanation will follow about what gaps can be bridged from the tools presented in this chapter and built upon in this project.

Chapter 3

Problem Analysis

The main purpose of this chapter is to analyse what makes up the PSA workflow, as well as an attempt to bridge a gap in the current tools and technologies that focus on supporting the full PSA workflow, which includes descriptive, predictive and prescriptive analytics. Moreover, a set of software requirements is listed based on which GPO-D is designed and developed. GPO-D should support full PSA workflow and improve on the productivity aspects that other tools and technologies lack (such as SolveDB). Lastly, the problem area is introduced from where the problem example (see Chapter 5) is taken and used to showcase the capabilities of GPO-D.

3.1 PSA Workflow

PSA workflow is used to describe a superset of tasks that belong to all phases of analytics, namely descriptive, predictive and prescriptive analytics [14]. More specifically, based on the Figure 3.1 a prescriptive analytics application is composed of:

- data collection and consolidation referred to as **data processing workflow** in this report
- creation of predictions referred to as **Machine Learning (ML) or ML workflow** in this report
- and set of steps composed of:
  - identification of alternative decisions and objectives
  - modelling and simulation of alternative decisions
  - selection of optimal decision
  - process analysis

  referred to as **mathematical optimization (MO) or MO workflow** in this report.

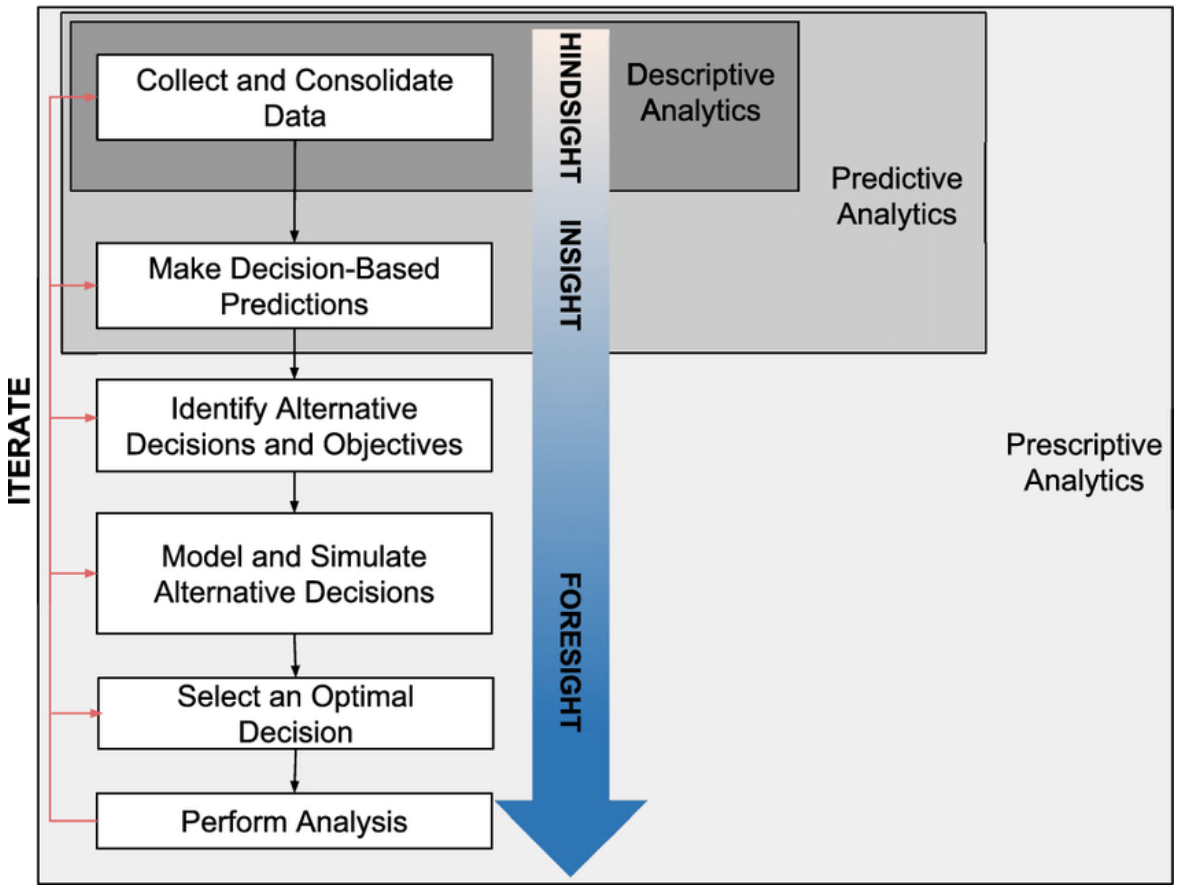

Figure 3.1: PSA workflow from Frazzetto et. al [14]

3.1.1 Expanding on ML Workflow

As can be seen in Figure 3.1, the predictive analytics phase (or ML workflow), which can be assumed to be a more complex chain of activities compared to data processing or MO workflow, is expanded for this thesis. Hence, CRISP-ML(Q), which stands for Cross-Industry Standard Process for Machine Learning with Quality assurance, has been employed in order to get an overview of activities that fall under the ML workflow. This development process model has been created with the idea of giving structure to ML projects and improving development efficiency and success of such applications [15] [16]. The actual phases of CRISP-ML(Q) are [16]:

- business and data understanding
- data preparation
- model engineering
- model evaluation

- model deployment
- model monitoring and maintenance

Furthermore, each phase has its set of tasks which will be listed in the following Section 3.1.2.

3.1.2 Adapted PSA Workflow

Based on the Figure 3.1 from Frazzetto et. al [14] paper and CRISP-ML process model [16], an adaptation of a more detailed PSA workflow has been created for this thesis (see Figure 3.2). The Figure 3.2 means to describe an iteration cycle (read from left to right) of the phases seen at the top row, together with a set of tasks below each of the phases not necessarily executed in top to bottom order with some also being optional in a PSA application. We will be referencing this adaptation of the PSA workflow whenever we refer to the adapted PSA workflow.

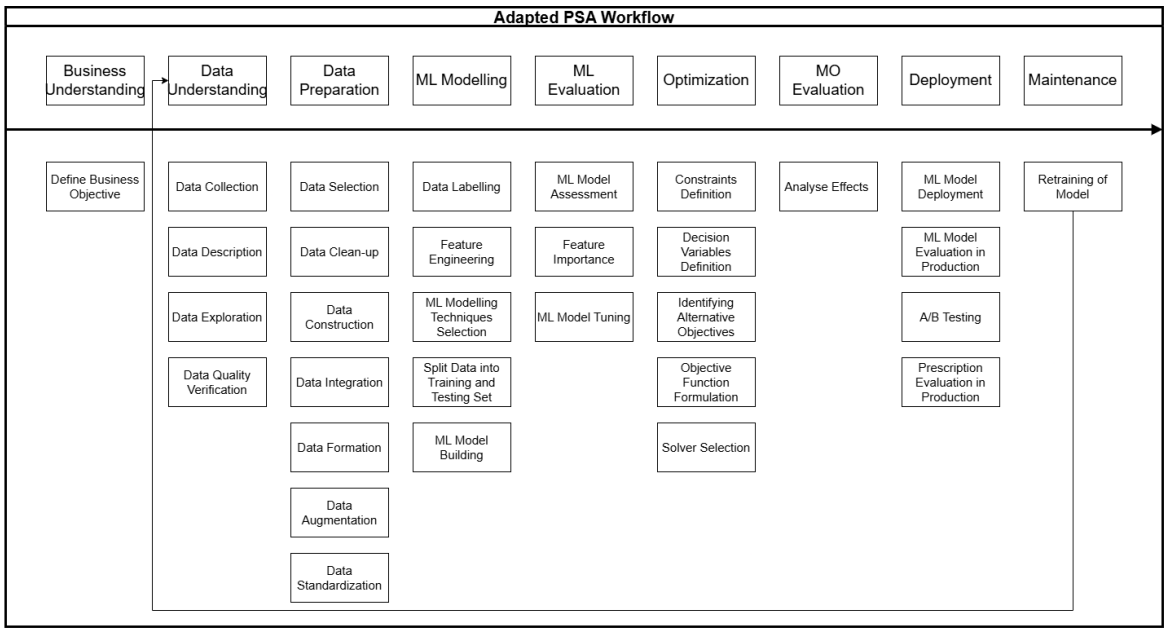

Figure 3.2: Adapted PSA workflow

For further referencing in this report, the phases with its tasks from Figure 3.2 has been grouped into:

- data processing workflow: data understanding, data preparation
- ML workflow: ML modelling, ML evaluation

- MO workflow: optimisation, MO evaluation

and where business understanding, deployment, and maintenance phases belong to their own categories.

3.2 Research Gap

Despite the increasing interest and research in data-driven decision support, currently available tools fall short in supporting the full PSA workflow effectively [14]. Such solutions tend to be fragmented, and often require the use of multiple different languages, technologies, tools, etc. As mentioned in the paper from Frazzetto et. al [14], the major limitations that have been identified are as follows:

1. **Fragmented workflow support**

   Most current tools and libraries are only built around descriptive and predictive analytics tasks [14, 3]. Prescriptive tasks are treated as a separate, disconnected step of the workflow. Therefore, users are forced to stitch together multiple such tools in order to implement the complete PSA workflow. Such ad hoc solutions have a big impact on users' productivity as they are now forced to context-switch between the different tools [14, 3].

   For example, systems such as BayesDB [17], GBOML [18], and SolveDB [6] are designed to only support one or two parts of the PSA workflow. BayesDB is used for predictive tasks, while GBOML and SolveDB are used for prescriptive tasks. Stitching these systems together would only lead to an ad hoc solution that is both error-prone and hard to maintain.

   Amongst the developments in the PSA area, SolveDB+ [7] has managed to rectify this limitation by extending SolveDB with additional features that provide support for predictive tasks on top of the existing optimisation problem-solving features. Likewise, the goal of this project is to develop a tool that can support more than just a single step in the PSA workflow. Ultimately, the defining feature would be the support for building complete PSA workflows without the user having to resort to additional out-of-scope tools.

2. **Lack of unified interface**

   Existing solutions often require multiple programming languages for the different stages of the PSA workflow [14]. That is tightly connected to the first limitation mentioned above, as there are only a few that fully support the entire workflow. That

not only increases development complexity but also makes it harder for business analysts and domain experts to learn to use it. Meanwhile, SolveDB+ manages to close that gap by unifying the predictive and prescriptive tasks within the same SQL-based system, such that users only have to learn the SQL language in order to do their job [7].

The goal of this project is to develop a tool that provides high-level, user-friendly abstractions that are based only on a single language, while still making sure that there is support for the entire PSA workflow. The idea is to have a unified interface in a language that is both popular and easy to learn by data scientists, e.g., Python. The SQL language used within SolveDB and SolveDB+ may be faster and more efficient, but it falls behind Python, for instance, in terms of flexibility, debugging capabilities, and more advanced analytics tasks that involve ML [19].

3. **Poor extensibility**

   Traditional analytics systems are often closed to extensions, i.e., they offer limited support for integrating new models, solvers, or algorithms. That is not a desirable quality because of how rapidly business environments evolve. On the other hand, modern systems like SolveDB+ allow users to extend the system if needed.

   Among the top priorities of this project is to design the tool in such a way that it can easily be extended to optimisation problems from the selected domain, or data source, or forecasting model, or solver, etc.

3.3 Target user-base

The target user base for this tool is the data scientist, who wants to use prescriptive analytics, but does not necessarily have the domain knowledge to implement a whole workflow, but they are comfortable with fetching data, using Python data frame libraries, and the predictive process, i.e. training and using machine learning models. The library should be also extendable, to other problems. This would be done by domain experts e.g., in energy sector.

Increasing Developer Productivity When making a workflow, like the battery-arbitrage from last semester [20], the user/developer is required to have knowledge in multiple fields. From the example, this includes: electrical engineering (or just battery engineering) for describing the battery dynamics, power-trading to understand what features a predictive model needs to make accurate predictions, and data-science for training the prediction model and using the different libraries.

We mainly focused on the Difference in Technical Knowledge like what data analysts have and domain knowledge. We base our assumptions on Gathani et al.(2024) [21], who conducted interviews with domain experts who lack the necessary skills to introduce data science or advanced data analytical methods into their workflows. The paper points out that data analysts lack this domain knowledge, too. This paper was mainly focused on what-if analysis, which can be considered a tool within the prescriptive analytics workflow.

3.4 Software Requirements

To guide the development of the tool, we define a structured set of functional (Table 3.1) and non-functional (Table 3.2) requirements. These requirements reflect both the features of the solution as well as broader goals of ensuring user-friendliness, extensibility, scalability, etc. They are categorised and prioritised using the MoSCoW method in order to clearly distinguish the core functionality from the potential future improvements.

3.4.1 Functional Requirements

The core of the tool is about enabling data scientists to define, model, and solve PSA problems in a structured and user-friendly manner.

Each functional requirement is oriented towards improving developer productivity, reducing manual work when defining and solving a PSA problem, and enabling extensions to other domain use cases.

ID	Requirement	Notes	MoSCoW
FR1	The tool should support an abstraction through which data scientists are able to represent/model problem domains that they are working in, as well as derive constraints and a possible objective function for the optimisation problem.	Using these abstractions, a data scientist must be able to model an MOP from the energy sector.	Must have
FR1.1	Objective function builder.	Have a function that builds objective functions from the domain specified. E.g. minimise power flow through the transmission lines.	Must have

ID	Requirement	Notes	MoSCoW
FR1.2	The abstractions should be extendable to other use-cases from the chosen sector.		Must have
FR2	The tool should support abstractions through which data scientists are able to simplify the data processing and ML workflows.	Similar to an idea of DAGs from Apache Airflow mentioned in chapter 2, and where data can be passed from one task to another one in the PSA workflow.	Must have
FR2.1	Partial automation of ML workflow.	Automate prediction model choice, feature engineering, or data pre-processing.	Should have
FR3	The tool should be able to store the models of the mathematical optimisation problems.	Similar to the idea of object-oriented modelling of optimisation problems in GBOML described in chapter 2. Store entire graphs in a persistent storage and be able to load them into memory on demand.	Should have
FR4	The tool should support the functionality of automatically choosing the (best) solver (as there are different types of solvers, such as CPLEX, GLPK, CBC, etc.) based on the optimisation problem type and then use that solver throughout the production cycle.	This function should only need to be run once when the problem implementation is finished. There should be a way to "remember" the best choice of a solver for a given problem.	Could have
FR5	The tool should support parallel execution of tasks in the PSA workflow.		Could have
FR6	The tool should automate the choice of a dataframe library based on the dataset size.	Have a list of dataframe libraries with some suitability parameters such as suitable dataset size, supported functionality, etc.	Won't have

ID	Requirement	Notes	MoSCoW
FR7	Translation functionality from GPO-D formulation into chosen Python optimisation libraries such as Pyomo, SciPy.		Won't have
FR7.1	Complex objective functions with weights.		Won't have
FR8	Adding a graphical user interface.	Drag and drop dags.	Won't have

Table 3.1: Functional requirements

3.4.2 Non-functional Requirements

Certain quality attributes have been a focus when developing the tool. It is crucial that the tool can support different problems and use cases, and that is achieved by ensuring that it is extendable. In addition to extensibility, the tool should still be able to handle problems of various sizes, in order to keep up with competitor tools.

ID	Requirement	Notes	MoSCoW
NFR1	The tool should increase developer productivity when working on PSA problems compared to other tools, such as the implementation of the same problem in a similar or same environment, e.g. in Python.	Productivity is measured in terms of effective lines of code and number of characters.	Must have
NFR2	Performance	GPO-D should be comparable to other tools in terms of execution speed and memory usage, where the same PSA problem can be solved.	Should have
NFR3	Scalability	GPO-D should be able to handle increasing problem sizes without compromising performance.	Should have

Table 3.2: Non-functional requirements

3.5 Problem Domain

Our tool will focus on the electric energy sector. This is because of our previous experience in the sector. We will mostly be focusing on energy flows in the grid, idealizing them to a simple mathematical problem, while abstracting away foundational details of electrical engineering concerns, we will only focus on balancing the production and consumption, as an example we will ignore frequency and voltage dynamics, this is partly due to our maximum 15 minute time resolution. Furthermore, we are not aiming for completely realistic problem parameters. The formulas themselves in Chapter 5 - Problem Example will be similar to those used in solving an actual problem with real data. However, the parameters for those formulas will not. As an example, the daily consumption of a household will be in the expected range, however, it might not reflect an actual consumption. This can be done, since we assume that the mathematical optimisers provide correct solutions to the given problems.

Chapter 4

Problem Statement

The main goal of this thesis is to develop a PSA tool in Python with developer productivity and user-friendliness in mind within the PSA field that lacks such tools, as discussed in Section Research Gap 3.2. Furthermore, the tool must support the full PSA workflow, as discussed in section PSA Workflow 3.1 while trying to retain the performance aspects such as execution speed in comparison with other Python implementations like CVXPY, Pyomo and GBOML. The development in this paper revolves around the following main question:

How can we build the GPO-D Python library that supports the full PSA workflow (which includes data processing, ML and math. optimisation) while trying to increase the developer productivity and user-friendliness, make the library extensible to multiple problems from the energy sector, and also keep up with the overall execution speed when compared to Python libraries like CVXPY, Pyomo or GBOML?

- RQ1: What abstraction(s) are needed in our tool to make the development of a full PSA solution simpler and more oriented towards developer productivity?
- RQ2: How do we make our abstraction(s) extensible in a way that developers can implement different PSA optimisation problems from the energy sector?
- RQ3: How does a problem example solution implemented in GPO-D compare to the implementations in CVXPY, Pyomo and GBOML in terms of performance?

To summarise, the goal of this paper is to deliver a functional prototype of the GPO-D Python library that implements a more complex version of the running example, which is the microgrid example, and describe how that is done. The main focus is on the top three requirements from the prioritised list below, although the system is designed with the rest of the requirements in mind to allow for easier future integration.

Prioritized list of "need to have" requirements:

1. FR1 - Abstractions for modelling Mathematical Optimisation Problems (MOPs)
2. FR2 - Abstractions for integrating the data processing and ML workflows
3. NFR1 - Developer productivity
4. NFR2 - Performance
5. NFR3 - Scalability

Chapter 5

Problem Example

We will now introduce an example that is implemented using the described formulas later in this chapter. All experiments provided in this thesis will be based on this. This example is essentially an upscaled version of the running example that was introduced in the beginning. This example will produce a 24-hour strategy based on weather forecasts and predicted data. This is not a replacement for a real-time control system.

5.1 *Microgrid* example

What is a microgrid?
The U.S. Department of Energy's Grid Deployment Office defines microgrids as a collection of consumers, producers and energy storage devices. The components of a microgrid are interconnected with a well-defined connection to the public grid. Thus, from the public grid's point of view, it acts as a single entity. The microgrid can provide resilience against failures on the public grid by being able to separate itself and satisfy the consumer's needs through local production. That is called an island mode. Microgrids can even sell back excess energy in some cases. [22]

The goal for microgrids is to avoid using electricity from the grid and try to maximise *self-consumption*, which is the idea of trying to satisfy local electricity needs with local production. We used residential microgrids as an example, since they can use different types of producers and the consumption curves may differ per consumer, because different households or other consumers may exhibit a specific consumption pattern. This gives us a nice opportunity to show that our tool can handle a diverse portfolio of consumers and producers.

5.1.1 Components of our example microgrid

Generally, microgrids consist of producers, consumers, and some type of energy storage systems. For our example, we will have:

- 50 homes with controllable solar panel installations (5 kW rated production capacity)
- a school with a controllable solar panel installation (25 kW rated production capacity)
- 1 wind turbine (1000 kW rated capacity)
- a grid connection in the DK1 grid area
- 3 battery storage systems with a capacity of 500 kWh, a round-trip efficiency of 0.9 and a charge/ discharge power of 500 kW.

The goal is to only use the public grid in situations where the locally produced energy does not satisfy the demand. In this case, the producers in our microgrid will have a production curve, but the production in the microgrid can be reduced if needed, this can be necessary in the scenario when the energy can not be exported and the batteries are charged, or in case the energy price is negative and exporting it would result in an extra cost.

5.1.2 Controlling solar panels

We will model controllable solar panels that will have a forecasted production curve, which will be based on the solar irradiance forecast from a meteorological source. The optimiser will be able to select a production capacity percentage between zero and the maximum of that curve, an example of this control can be seen in Figure 5.1. We are aware that this type of central controllability is not typical for rooftop solar. For example, it is possible to set up electricity export limiting at a household level [23], so we are assuming that it could be done at a community level.

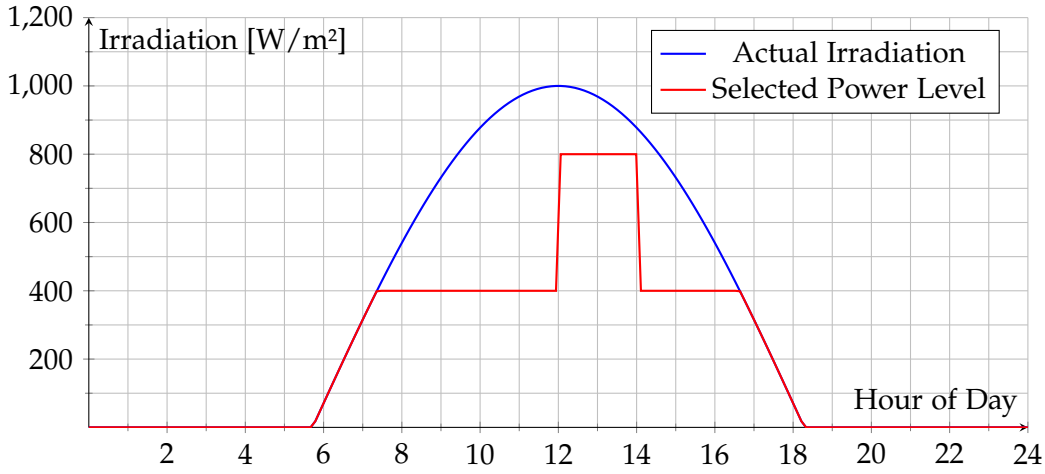

Figure 5.1: Example irradiation curve

5.1.3 Mathematical formulation

Input data

The input data for this example will be the forecasted irradiation and wind speeds. Both of them will be of length T, which is the number of timesteps for the problem. $t = 0, 1, \ldots, T$ denotes the timesteps.

Producers

The energy of the producers is given by the following equation:

$$E_t \quad [\text{kWh}] = P_t \cdot \frac{1}{l}$$

where:

- P_t [kW] is the power of the producers for time step t.
- l [h] is the length modifier for the time step. If we are working with hourly time steps, this would be 1. If we are working with 15-minute time steps, this would be 4.

Solar panels

The power of the solar panels is given by the following:

$$P_t = c_t \cdot P_{rated} \cdot Q_t$$

where:

- $0 \leq c_t \leq 1, c_t \in \mathbb{R}$ is the control variable for the optimiser, for time step t. This is a decision variable for the optimiser.
- P_{rated} [kW] is the rated power of the solar panels for an irradiance of 1 kW/m^2. [1]
- Q_t [kW/m^2] is the solar irradiance, for time step t.

Wind turbine

For the wind turbine, we rely on a simplified model, which can be seen in Figure 5.2 [25]:

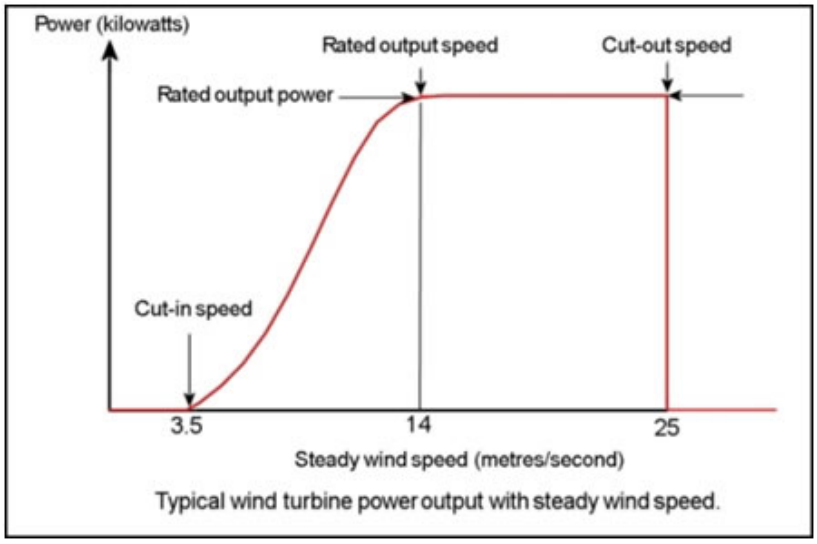

Figure 5.2: Wind turbine power generation, depending on the wind speed, reproduced from [25]

$$P_t(v) = \begin{cases} 0, & v_t < v_{\text{cut-in}} \\ P_{\text{rated}} \left(\dfrac{v_t - v_{\text{cut-in}}}{v_{\text{rated}} - v_{\text{cut-in}}} \right)^3, & v_{\text{cut-in}} \leq v_t < v_{\text{rated}} \\ P_{\text{rated}}, & v_{\text{rated}} \leq v_t < v_{\text{cut-out}} \\ 0, & v_t \geq v_{\text{cut-out}} \end{cases}$$

where:

- $P_t(v)$ [kW] is the produced power at time step t for a given wind speed v.
- P_{rated} [kW] is the rated power of the wind turbine.

[1] This is a simplified model, we assume that the efficiencies and variables, based on installation, are already included in this number. Generally, in photovoltaic systems, the standard test conditions are $1000 W/m^2$.[24]

- v_{cut-in} [m/s] is the cut-in wind speed for the turbine. This is the minimum wind speed, where the turbine can produce power.
- $v_{cut-out}$ [m/s] is the cut-out wind speed for the turbine. This is the maximum wind speed until the turbine produces power, if the wind speed is greater than this, then the turbine shuts off for safety reasons.
- v_t [m/s] is the predicted average wind speed at time step t.

Consumers

For the consumers, we used an online available dataset for past household electricity consumption [26]. We will use an automated machine learning tool to forecast future consumption. Generally the consumption data will be an array $E_t = [-e_0, -e_1 \cdots - e_T]$, with all values given in *kWh*. We did not find a usable dataset for the consumption of the school. Instead, we used a scaled-up consumption curve from the household dataset. This scaling will be termed the consumption scaling factor f_c and the resulting consumption of the school will be given by the equation $E_t = [-f_c e_0, -f_c e_1 \cdots - f_c e_t]$. We are aware that this is not a realistic consumption curve for the school.

Batteries

We modified the formulation from the battery arbitrage example from our last semester's report [20]. We changed the units and replaced the power values with energy values, which is due to the changeable length of the timesteps.

Variables

- $E_{t,c} \geq 0$ [kW] be the charged energy at time t,
- $E_{t,d} \geq 0$ [kW] be the discharged energy at time t

Decision Variables

- $s_t \geq 0$ [kWh] be the state of charge (SOC) at time t

Constraints

1. **Initial and Final State of Charge**

$$s_1 = 0, \quad s_T = 0$$

2. **State of Charge Dynamics**

$$s_{t,i} = \begin{cases} \eta E_{t,c} - \frac{1}{\eta} E_{t,d} & \text{if } t = 1, \\ s_{t-1} + \eta E_{t,c} - \frac{1}{\eta} E_{t,d} & \text{if } t > 1, \end{cases}$$

where:

- η is the efficiency of the battery ($0 < \eta \leq 1$).

3. **Energy and Capacity Limits for Each Battery**

$$0 \leq E_{t,c} \leq Pmax_c \cdot \frac{1}{l}, \quad 0 \leq E_{t,d} \leq Pmax_d \cdot \frac{1}{l}, \quad 0 \leq s_t \leq C$$

where:

- $Pmax_c$ [kW] is the maximum charging power of the battery,
- $Pmax_d$ [kW] is the maximum discharging power of the battery,
- C [kWh] is the capacity of battery.
- l [h] is the length modifier for the timestep. If we are working with hourly timesteps, this would be 1. If we are working with 15-minute timesteps, this would be 4.

The resulting energy into the microgrid from the battery would be defined by:

$$E_{battery,t} = E_{t,d} - E_{t,c}$$

where:

- $E_{t,d}$ [kWh] is the discharged energy at timestep t.
- $E_{t,c}$ [kWh] is the charged energy at timestep t.

Energy flow per timestep

The resulting energy flow for each time step would be calculated by summing up all the components' energy flows for each timestep. This can be done since the values would be signed and generally represent production into the grid with a positive number and consumption from the grid with a negative number:

$$E_t = \sum_{d=0}^{N} E_{d,t}$$

where:

- $E_{d,t}$ [kWh] is the energy produced or consumed by component d at time step t for the system. This would include all consumers, producers and the battery.
- N is the number of components in the given microgrid.

Objective Function

We defined two objective functions with different goals:

1. To minimise energy flow in and out of the microgrid, i.e. to minimize the usage of the grid.

$$\arg\min_{C,S} \sum_{t=0}^{T} |E_t|$$

where

- E_t [kWh] is the energy produced or consumed by the microgrid at time step t for the system.
- T is the optimisation's time horizon, i.e. the number of timesteps.
- $C = \{c_0, c_1, c_2, \dots c_T\}$ and $S = \{s_0, s_1, s_2, \dots s_T\}$ are sets for all the decision variables defined by the previous components. Both with a length of T.

2. To maximise the profit from the sold electricity, i.e. to minimise the cost of the energy. This objective function is taken from our last semester's project. [20]

$$\arg\max_{C,S} \sum_{t=0}^{T} E_t p_t$$

- E_t [kWh] is the energy produced or consumed by the microgrid at time step t for the system.
- T is the optimisation's time horizon, i.e. the number of timesteps.
- $C = \{c_0, c_1, c_2, \dots c_T\}$ and $S = \{s_0, s_1, s_2, \dots s_T\}$ are sets for all the decision variables defined by the previous components. Both with a length of T.
- p_t [EUR/kWh] is the day-ahead price of the electricity at time t. [2]

[2]Generally day-ahead prices would be given in EUR/MWh. But we assume that the conversion to EUR/kWh would be done in the data preparation process. In this case, it would be done by dividing the prices by 1000.

Chapter 6

Design & Implementation

This chapter will present the design of GPO-D. We first start by looking at the high-level architecture overview of the GPO-D and how it is connected to our adapted PSA workflow. Secondly, the technologies and libraries being used to build the tool are explained. Lastly, two design concepts are explained, *dataflows* and graph modelling, which are proposed in order to help simplify the development and modelling of full PSA problems and improve developers' productivity when implementing solutions to those problems.

6.1 High-Level Overview

Figure 6.1 illustrates a high-level architecture of GPO-D, which operates in three main stages.

The first stage involves defining a specific optimisation problem by modelling it as a graph. This is done through the implementation of domain-specific classes such as solar panels, batteries, or consumers, which inherit from base classes provided by GPO-D. These domain-specific classes, referred to as *domain nodes* (or simply *nodes* in most contexts), are associated with a central structure that represents the whole optimisation problem.

In the second stage, for each *node* we can optionally define a workflow (referred to as *dataflow*) that is responsible for collecting and processing the data needed by the *nodes*. These *dataflows* may include data retrieval tasks (e.g., fetching weather forecasts) or ML tasks (e.g., predicting household load/consumption).

The third stage begins when all *dataflows* are executed and have completed. At this point, the tool automatically gathers constraints, decision variables, and cost functions from each *node*. These are then compiled into an optimisation model and run using a selected solver. The resulting solution to the optimisation problem is then returned to the user.

6.1.1 Segmentation of workflows

This segmentation between the *dataflow* and *(domain) nodes* is done to separate the work of domain experts and data scientists. The idea is that a *dataflows* are developed by a data scientist, while the *nodes* and their related constraints are implemented by an energy domain expert. And the final integration and testing of these components can be done by any developer who is familiar with the basics of object-oriented programming.

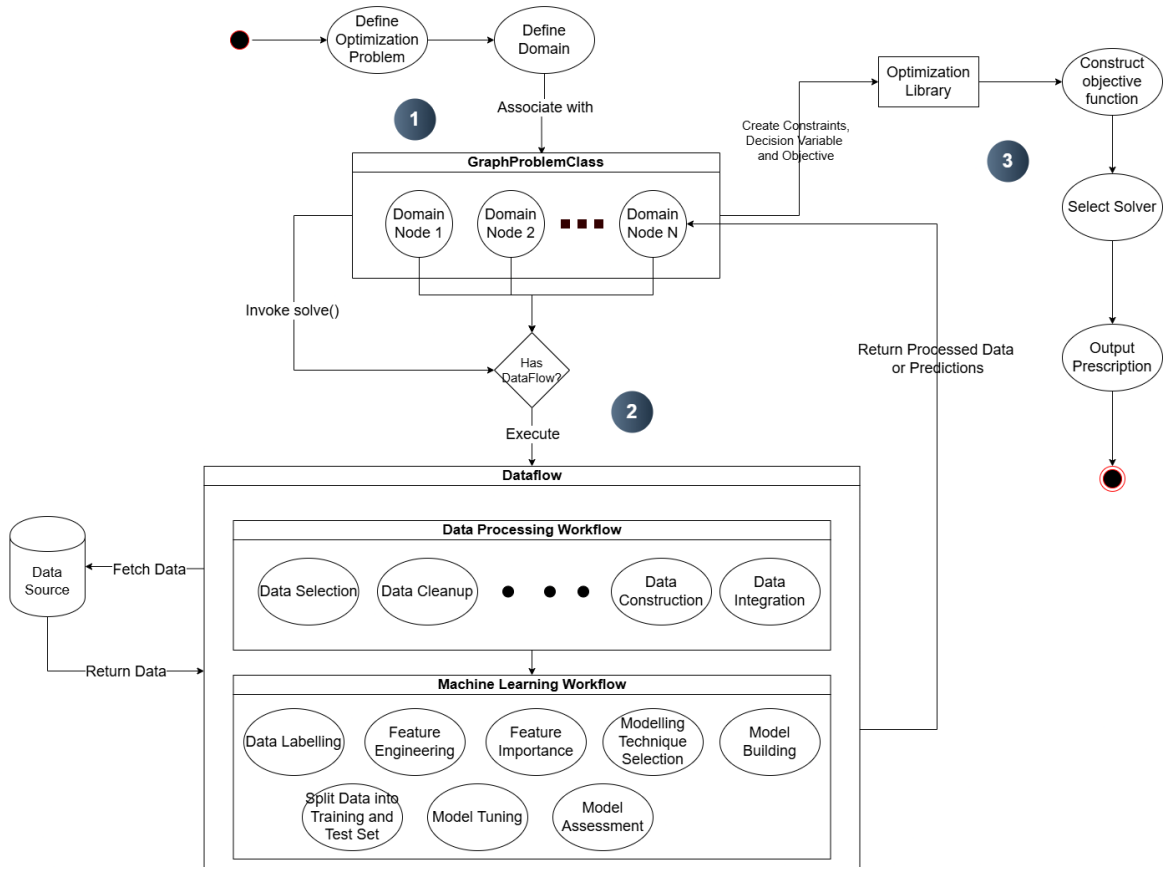

Figure 6.1: High Level Overview of GPO-D

6.2 Adapted PSA Workflow Usage

We set out to describe the PSA workflow in section Adapted PSA Workflow 3.1.2. Figure 6.2 represents a coloured version of this workflow depicting which tasks are ignored within the tool (grey) due to time limitations, defined by the user/developer of our tool (red), partially automated by the tool (orange), and fully automated by the tool (green).

Since making a generic tool capable of modelling and solving any business use case is

difficult, we are leaving the *Business Understanding* completely in the hands of the user. This involves both modelling the domain, i.e., through the use of the graph representations described in section Graph Modelling 6.5, and also defining the business objective, which can be very specific to each optimisation problem. Furthermore, the data collection is not the focus of this tool, thus, it can be assumed that data for our purposes is already stored in a database or CSV file. For the most part, tasks from *Data Understanding* and *Data Preparation* phases are mostly done by the user based on the problem at hand, due to challenges in automating it (one of the few relatively easier automations can be done in data cleaning).

From the *ML Modelling* and *ML Evaluation* phases, which are not the focus of GPO-D, most tasks can be partially or fully automated such as feature engineering (e.g., creating the sine and cosine transformations to handle cyclic variables), feature importance (e.g., trying out different combinations of features until the best one is found, one with least error), ML modelling technique selection or ML model tuning.

As the integration of the *Optimisation* part with the rest of the process is the main goal of this tool, the definition of the business constraints, decision variables, and the formulation of the objective function is automated to some degree. The user selects or defines the objective function and specifies constraints and decision variables, but that can be done through the intuitive graph concept explained in section 6.5. Then, the constraints, decision variables and objectives will be extracted from the graph itself automatically. The selection of the solver for the optimisation problem will also be automatically done, but we will also give the freedom of selecting a specific one, if the user wishes to do so.

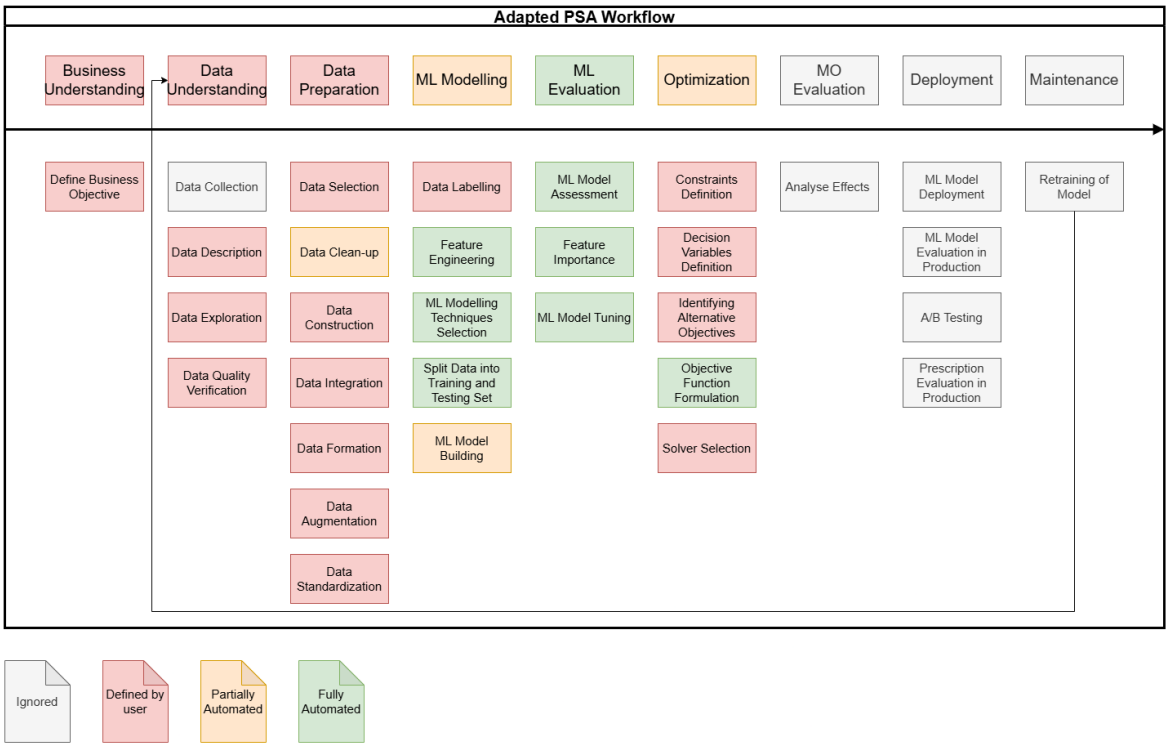

Figure 6.2: Adapted PSA workflow coloured

6.3 Technology Stack

In this section, we outline the primary technologies and libraries used in the implementation of GPO-D. The selected stack has support for each of the steps in the PSA workflow.

6.3.1 Application Programming Language

The primary programming language used for GPO-D is Python. As the main target user of GPO-D is the data scientist, Python is a suitable option due to being widely adopted in the data science community, as well as having a vast ecosystem of libraries and active community support. The flexibility and interoperability that it offers, as well as the ability to seamlessly integrate data processing, machine learning, and optimisation components into a single solution, further enable the development of an end-to-end PSA workflow [27].

6.3.2 Data Processing

For data processing and manipulation, we use the *pandas* [28] Python library. Pandas provides data structures like DataFrames that can be used for handling structured data. Pandas in general is extensively used for data cleaning, transformation, and data analysis.

Furthermore, it offers high-performance data manipulation when working with small to moderate-sized datasets. In the problem examples described in this thesis, all datasets are of a smaller size, therefore, pandas proves sufficient in the implementation of GPO-D. [29]

6.3.3 Machine Learning

For the machine learning component of GPO-D, we use *PyCaret*, which is an open-source, low-code ML Python library [30]. It is used for automating ML workflows, therefore, it is considered as part of the AutoML [31] family of tools. It considerably simplifies the ML process by automating steps such as data preprocessing, model comparison and selection, model training, and "hyperparemeter" tuning [30]. It serves as an abstraction layer or a wrapper over common ML frameworks (such as XGBoost [32], CatBoost [33], LightGBM [34]), hence allowing for rapid prototyping and model selection, making it particularly useful for predictive tasks in the PSA workflow. We used this library to automate the ML workflow for GPO-D.

6.3.4 Mathematical Optimization

For the optimisation phase of the PSA workflow, we use *cvxpy* [35]. cvxpy is a Python-embedded modelling language used for expressing and solving convex optimisation problems [35]. Firstly, it enables users to express MOPs in a readable way that follows the math behind the problem itself [35]. Secondly, cvxpy supports a wide range of solvers (such as OSQP, GUROBI, CBC, etc.) that can be used for linear and quadratic programming or constrained optimisation problems. The simple syntax of cvxpy, and the large support of solvers are reasons why it is a suitable choice for GPO-D. We used it as a high-level interface for the solvers.

6.4 Dataflow

The core prerequisite when solving an MOP is access to accurate and relevant input data, as it forms the basis for all decision-making. To reliably obtain such data, we need a well-established and structured process, referred to as a *dataflow*. The *dataflow* defines how the input data necessary for solving an MOP is acquired and prepared. This includes specifying the source of the data, the component of the MOP for which the data is relevant, and the mechanism by which the data is fetched or retrieved, and subsequently loaded into memory or persistent storage for use during optimisation.

6.4.1 Dataflow Definition

The different node types (e.g., *SolarPanel*, *Consumer*) in the MOP graph often require distinct workflows to process and prepare the data.

For example, if we consider solar panel production vs. household consumption. These can use different data sources, different prediction models, and the preparational steps can be different. Even when we consider different models of solar panels, or when operated by different companies, the data format can be different, or the source of the data can be different, thus needing a different workflow to convert the data for the optimiser. However, solar panels, which are similar in workflow but have only parametric differences, i.e. different locations or different rated power output, can be processed by the same process.

To alleviate this issue, we adopted the DAG concept that Apache Airflow [36] uses. Each node type is assigned a different workflow for fetching and preparing the necessary data. In the example in Figure 6.3, the *SolarPanel* nodes have an Apache-like DAG described to fetch and process the required data. We call these DAGs - *dataflows*, and each step in the *dataflow* (e.g., *GetSolarPanelData*, *PreProcessData*) - a *task*.

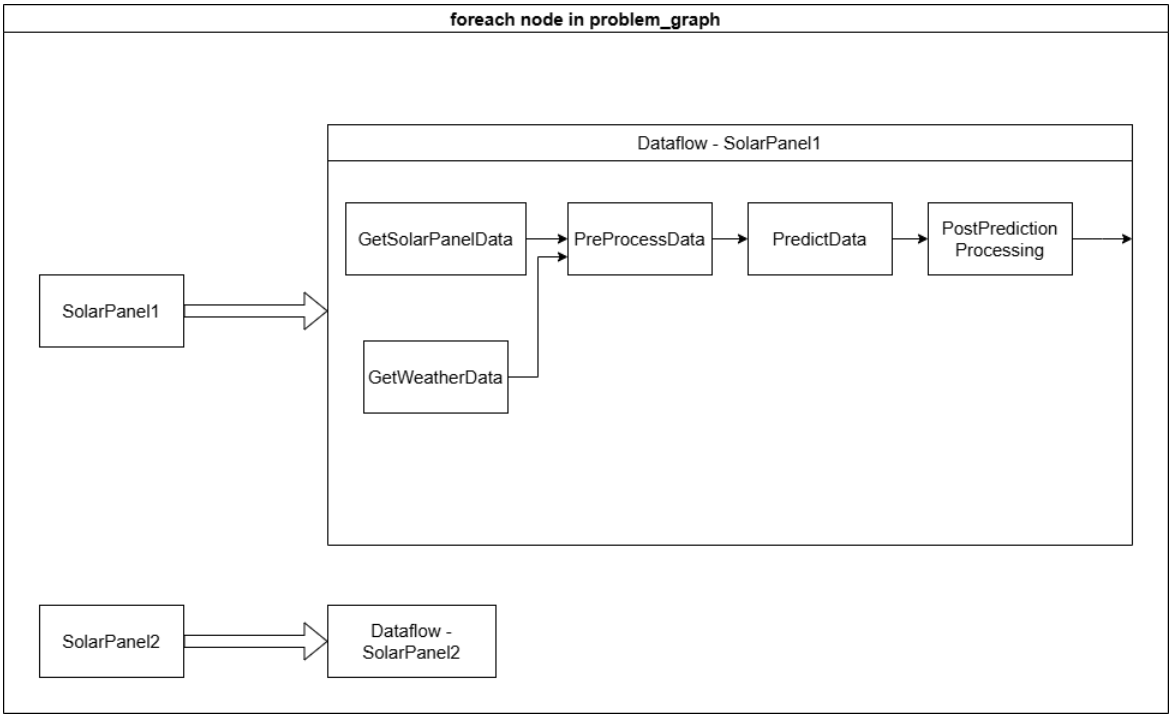

Figure 6.3: Dataflow definition

DataFlowTask

A *DataFlowTask* (in short *task*) represents a single unit of work in the *dataflow*. This can be anything from fetching data from various sources, processing it, or running an ML model. Each *task* can depend on other *tasks*, which form a graph-like structure (as seen in Figure 6.3).

Listing 6.1 shows the implementation of the *DataFlowTask* class. Each *task* is initialised with a unique name, a set of parameters (e.g., latitude and longitude for a wind turbine), and a flag indicating whether it is the final task in the *dataflow*. It also maintains a list of dependencies, i.e., other *tasks* that must be executed before this one.

The execution logic of a *task* is captured in the *run* method. When a task is run, it first recursively executes all dependencies, which ensures that all prerequisite data is available. The results from the dependencies are collected and passed to the *task*'s *process* method. The *process* method contains the logic to perform the actual computation, and the results of the computation are then stored within the *task* for later use by other *tasks* or the optimisation problem.

```
1  class DataflowTask:
2      def __init__(self, name: str, parameters = {}, final = False):
3          self.id = uuid.uuid4()
4          self.name = name
5          self._dependencies = []
6          self._results = {}
7          self._final = final
8          self._parameters = parameters
9
10     def add_dependency(self, task):
11         self._dependencies.append(task)
12
13     def run(self):
14         input_dfs = {}
15         for task in self._dependencies:
16             task.run()
17             input_dfs.update(task.get_results())
18         self.process(input_dfs)
19         self._results = input_dfs
20
21     def process(self, dfs: Dict[str, pd.DataFrame])
22     -> Dict[str, pd.DataFrame]:
23         raise NotImplementedError("Subclasses must implement this method")
24
25     def get_results(self) -> Dict[str, pd.DataFrame]:
26         return self._results
```

Listing 6.1: DataFlowTask implementation

A notable feature of *DataFlowTask* is the overloading of the » operator (see Listing 6.2), which is used for chaining of *tasks*. When *task1 » task2* is written, the overloaded method will add *task1* as a dependency of *task2*. This syntactic sugar is added to make the construction of dataflow graphs both more readable and concise.

```
# Overload >> operator for dependency chaining
def __rshift__(self, task):
    task.add_dependency(self)
    return task
```

Listing 6.2: Overloaded » operator implementation

Task Types

To support different types of operations within a *dataflow*, GPO-D defines several specialised subclasses of *DataflowTask*. Some of these include:

- **DataFetchingTask**: An abstract base class for *tasks* that retrieve data from various sources, such as files, databases, or APIs. There are concrete implementations like *DataFetchingFromFileTask* and *DataFetchingFromDBTask* that provide the logic for retrieving data from specific data sources.
- **DataProcessingTask**: A *task* that applies a user-defined function to its input data. This allows for any arbitrary data transformations to be encapsulated as *tasks*.

Each specialized *task* must implement the *process* method from *DataFlowTask* to define its specific behaviour. For example, a *DataFetchingFromFileTask* reads CSV files, while *DataProcessingTask* might merge, filter, or do other operations on dataframes, depending on what the user-defined function is. Listing 6.3 shows the implementation of the two types of *tasks*.

```
class DataFetchingFromFileTask(DataFetchingTask):
    def fetch_data(self) -> pd.DataFrame:
        return pd.read_csv(self.source)

    def process(self, dfs: Dict[str, pd.DataFrame]):
        dfs[self.name] = self.fetch_data()

class DataProcessingTask(DataflowTask):
    def __init__(self, name, process_func, parameters={}, final=False):
        super().__init__(name, parameters, final)
        self.process_func = process_func
```

```
12
13      def process(self, dfs: Dict[str, pd.DataFrame])
14      -> Dict[str, pd.DataFrame]:
15          return self.process_func(dfs, self._parameters)
```

Listing 6.3: DataflowTask types examples

Building a *Dataflow*

A *Dataflow* manages a list of *tasks* and their dependencies for a specific node (e.g., a household, solar panel, etc.). As shown in Listing 6.4, the class provides methods to add *tasks* (*task* method), retrieve them by name (overloaded *[]* operator), and execute the entire *dataflow* (*execute* method). The *Dataflow* class ensures that *tasks* are executed both in the correct order and also respecting all dependencies. The results are then collected and made available for further processing or analysis.

```
1  class Dataflow:
2      def __init__(self, name, object) -> None:
3          self.name = name
4          self.object = object
5          self.tasks = {}
6
7      def task(self, name, task_node_type: DataflowTask=None, *args, **kwargs)
8      -> DataflowTask:
9          if name in self.tasks:
10             return self.tasks[name]
11         else:
12             if task_node_type is None:
13                 self.tasks[name] = DataProcessingTask(name, *args, **kwargs)
14             else:
15                 self.tasks[name] = task_node_type(name, *args, **kwargs)
16             return self.tasks[name]
17
18     def execute(self) -> None:
19         final_task = self.get_final_task()
20         if final_task is not None:
21             final_task.run()
22             self.results = final_task.get_results()
23
24     # overload [] operator
25     def __getitem__(self, name: str) -> DataflowTask:
```

```
26          return self.task(name)
```

Listing 6.4: Dataflow class implementation

Managing *Dataflows*

As the number of *dataflows* increases relative to the number of nodes, managing them efficiently becomes cumbersome. For that reason, we introduce the *DataFlowManager* singleton class, which has the responsibility of managing and coordinating all *dataflows* within the application. As shown in Listing 6.5, the *DataFlowManager* keeps track of all *dataflows* for different *nodes* in the optimisation problem graph definition. It ensures that each *node* is associated with a unique *dataflow* and that repeated requests for the same *node* return the same dataflow instance (*new_dataflow* method). Furthermore, it provides methods to execute all registered *dataflows* (*execute* method), and to retrieve results from *tasks* within a *node*'s *dataflow* (*get_data* method).

```
1  class DataflowManager:
2      def __init__(self) -> None:
3          self.dataflows = {}
4
5      def new_dataflow(self, object, dataflow = None) -> Dataflow:
6          if object.name in self.dataflows:
7              return self.dataflows[object.name]
8          if dataflow is None:
9              dataflow = Dataflow(object.name, object)
10         self.dataflows[object.name] = dataflow
11         return dataflow
12
13     def execute(self) -> None:
14         for dataflow in self.dataflows.values():
15             dataflow.execute()
16
17     def get_data(self, object, task_name):
18         dataflow = self.dataflows[object.name]
19         return dataflow.get_data(task_name)
```

Listing 6.5: DataflowManager class implementation

The following Listing 6.6 demonstrates how dataflows are constructed and executed for the components in the running example, specifically for a household consumer and a solar panel. First, a new *task* named "*gen_consumption*" is added to the household's *dataflow*. The *task* is of type *DataProcessingTask* and uses a user-defined function for generating the

household's energy consumption data. Similarly, for the solar panel, two *tasks* are defined. Since the second *task* of generating solar panel production data depends on solar irradiation data from the first *task*, that dependency is ensured by *task1 » task2* chaining operation. Finally, the *execute* method of the *DataflowManager* is called, which triggers the execution of all registered dataflows in the system. The results of the *tasks* flagged as "*final*", are collected and made available for further use in the optimization process.

```
household.dataflow.task(
    "gen_consumption",
    DataProcessingTask,
    process_func=predict_consumer_data,
    parameters={"hours": 24, "model_name":"consumer_model"},
    final=True
)

task1 = solar_panel.dataflow.task(
    "get_solar_data",
    DataProcessingTask,
    process_func=get_irradiation_data,
    parameters={"latitude": 57.0488, "longitude": 9.9217}
)
task2 = solar_panel.dataflow.task(
    "gen_solar_data",
    DataProcessingTask,
    process_func=generate_solar_panel_data,
    parameters={"rated_power": 5},
    final=True
)
task1 >> task2

self._dataflow_manager.execute()
```

Listing 6.6: Example *dataflow* for household running example

6.5 Graph Modelling

The core idea behind making a graph modelling system for optimisation constraints is to avoid using complex equations to represent connections in the electrical grid.

To make the development of prescriptive analytics workflows easier, we introduce a modular library that can model MOP using graph structures. The core of this concept revolves

around abstracting different parts of problem domains as graph-based models, thus allowing for flexible definition, manipulation, and solution of complex optimisation tasks.

The design is represented through abstractions, which include a set of core base classes. In addition to that, there is a set of specialised classes that capture the intricacies of a MOP domain (e.g., battery).

6.5.1 Design Principles and Considerations

When coming up with the graph modelling concept, we considered multiple design principles and we emphasised the following:

- **Separation of concerns:** The idea is that each domain for a MOP has its own set of specialised classes, all of which inherit from a shared set of generic base classes. The generic classes handle the core optimisation logic (i.e., collecting constraints, defining the objective, solving the problem), while domain-specific logic is delegated to the subclasses. Inheriting from the same base class ensures consistency, as all subclasses follow a common structure and expose a unified interface.
- **Extensibility:** New types of nodes and optimisation domains can be added with minimal changes to the existing architecture by simply extending the core base classes.
- **Reusability:** Code duplication and boiler-plate code are minimised through the usage of the generic base classes, which serve as universal building blocks for any MOP and domain.
- **User Productivity:** By introducing built-in high-level abstractions, such as *Energy Storage Unit*, *Producer*, etc., the user doesn't have to be an expert in mathematical modelling or know about low-level solver APIs. These domain objects hide the complexity of variables, constraints, and cost functions. Furthermore, due to the modular design of the library, users can incrementally build optimisation models by adding nodes, without the need to redefine the entire problem again.

6.5.2 Core Abstractions

Node

The concept of a *Node* reflects the main building block of the library. Each node is tied to a *GraphProblemClass* and contains problem-specific data such as constraints, cost functions, and variables. The nodes are modular and can be reused by creating copies of them, therefore allowing the user to scale up the complexity of their optimisation problem by adding more nodes.

GraphProblemClass

GraphProblemClass serves as a central container for the entire optimisation problem, maintaining a list of *Node* objects that encapsulate individual components of the optimisation problem, and handles the compilation and solution of the complete problem. This abstraction allows for the flexible construction of optimisation problems as undirected graphs, where the nodes are responsible for defining constraints, decision variables and cost functions. The flow of electricity will be represented by the sign of that number, a positive sign represents production of electricity, while a negative sign represents consumption of electricity.

Listing 6.7 shows a part of the implementation of *GraphProblemClass*. The class maintains a list of nodes (line 5), and provides the *add_nodes* method to append new nodes (lines 9-10). Furthermore, *GraphProblemClass* defines methods for collecting cost terms (*collect_costs*) and constraints (*collect_constraints*) from each node, which enables a modular construction of the complete optimisation problem. Lastly, the provided *fetch_dataflows* method ensures that each node's associated dataflow (if any) is added to the shared *DataflowManager* instance. This allows those dataflows to be executed prior to constructing and solving the optimisation problem.

```
1  class GraphProblemClass():
2      def __init__(self, name, time_length=24):
3          self.name = name
4          self.time_length = time_length
5          self._nodes = []
6          self._selector = None
7          self._dataflow_manager = DataflowManager.getInstance()
8
9      def add_nodes(self, nodes):
10         self._nodes.extend(nodes)
11
12     def fetch_dataflows(self):
13         for node in self._nodes:
14             if hasattr(node, "dataflow") and node.dataflow is not None:
15                 self._dataflow_manager.new_dataflow(node, node.dataflow)
16
17     def collect_costs(self):
18         return cp.sum([node.cost for node in self._nodes])
19
20     def collect_constraints(self, t):
21         constraints = []
```

```
22          for node in self._nodes:
23              constraints.extend(node.constraints(t))
24          return constraints
```

Listing 6.7: GraphProblemClass implementation

Listing 6.8 presents the core function of the *GraphProblemClass*, namely the *solve* method. This method orchestrates the entire workflow. It first registers and executes any associated dataflows for the nodes, then builds the objective function based on the user's specification (e.g., minimising or maximising a cost). Afterwards, it collects all time-indexed constraints from each node. Finally, it builds and solves the optimisation problem using a specified solver backend (e.g., GUROBI, OSQP).

```
1  def solve(self, solver=cp.GUROBI, objective: str = "minimize",
2      value: str = "cost"):
3
4      # Fetch dataflows from nodes and execute them
5      self.fetch_dataflows()
6      self._dataflow_manager.execute()
7
8      # Build the objective function
9      self._selector = Selector(self._nodes)
10     if objective == "minimize":
11         objective = cp.Minimize(cp.sum(self._selector.get(value)))
12     if objective == "maximize":
13         objective = cp.Maximize(cp.sum(self._selector.get(value)))
14
15     # Collect all constraints from the nodes
16     constraints = []
17     for t in range(self.time_length):
18         constraints.extend(self.collect_constraints(t))
19
20     # Build and solve the optimization problem
21     problem = cp.Problem(objective, constraints)
22     problem.solve(solver=solver)
```

Listing 6.8: GraphProblemClass *solve* function implementation

In Listing 6.9, we demonstrate how to define the single household running example using the *GraphProblemClass*. We instantiate the *GraphProblemClass*, define all relevant nodes (e.g., battery, solar panel, consumer, and metering point), register any required dataflows, and finally get the solution of the complete problem by invoking the *solve* function of the

GraphProblemClass.

```
# Create the problem instance
problem = GraphProblemClass(name="household_energy", time_length=24)

# Create and add nodes
battery = Battery(...)
pv = SolarPanel(...)
household = Consumer(...)
grid = MeteringPoint(...)

household.dataflow.task(...)

problem.add_nodes([battery, pv, household, grid])

# Solve the problem
problem.solve(objective="minimize", value="cost")
```

Listing 6.9: Composing the optimization problem using GraphProblemClass

6.5.3 Objective Function Constructor

The objective function constructor for the *GraphProblemClass* is designed to provide a flexible interface for defining optimisation objectives across a variety of problem types. The core idea is to allow users to specify an objective function by selecting and aggregating relevant variables from nodes in the graph. Users can then specify whether the selected variables should be minimised or maximised, which enables the reuse of the same logic across different problem domains.

For example, the objective of maximising self-consumption can be formulated as minimising the power flow through the *MeteringPoint* component of our running example. Another example could be maximising profit over the whole graph, which translates to minimising the sum of costs over the optimiser's time horizon.

The selection and aggregation of relevant variables from nodes is achieved by introducing a helper class called *Selector* (see Listing 6.10), which supports filtering nodes in the graph by type (using *of_type*), by attribute (using *where*), or other criteria. Once a *Selector* object is initialised, it is assigned to a *GraphProblemClass* and it is scoped to all nodes in the graph by default. The user can then filter this set and aggregate the selected variables to form a single scalar objective expression using the *get* method. Finally, the resulting objective expression is then used in the optimisation problem, either with a minimise or maximise

operation, depending on the user's initial choice.

```
class Selector:
    def __init__(self, items):
        self.items = items

    # Filters items by instance type
    def of_type(self, type_or_types):
        ...

    # Filters items based on the value of an attribute.
    def where(self, attr, value):
        ...

    # Filters list of attribute values for each item
    def get(self, attr):
        return [item.get_attr(attr) for item in self.items]
```

Listing 6.10: Implementation of the *Selector* helper class

Listing 6.11 demonstrates how the *Selector* can be used to build an objective. This code example can work for the household running example, since the *"cost"* attribute of a *MeteringPoint* node can be defined to return an expression of the sum of energy imported from the grid (*return cp.sum(self.energy_import)*) Consequently, the resulting expression can be minimized or maximized, to form the objective.

```
GraphProblemClass:
    self._nodes = []
    self._selector = None
    ...

    self._selector = Selector(self._nodes)
    objective = cp.Minimize(cp.sum(self._selector.get("cost")))
```

Listing 6.11: Using *Selector* to build an objective

The motivation behind the objective function constructor is to support a broad range of problems with some simple building blocks and with minimal user effort. The user should be able to define the objective using simple, composable filters rather than manually aggregating variables or writing hard-coded solver-specific or problem-specific code.

6.5.4 Connection handling

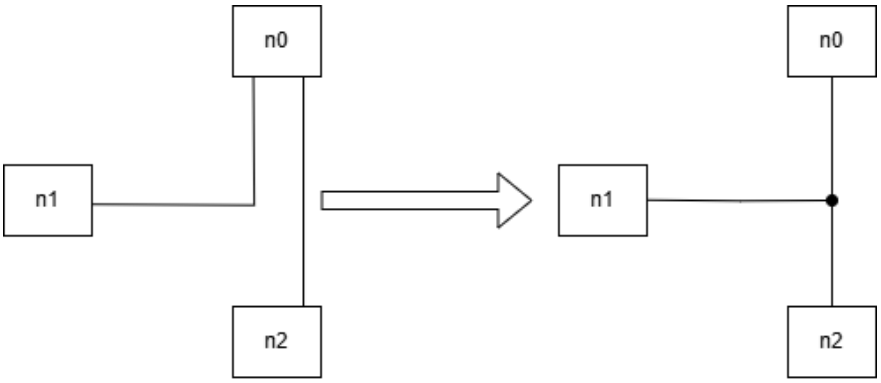

Figure 6.4: Connection conversion

This modelling supports many-to-many connections between the nodes in the constraint modelling graph. These are converted to have a logical common connection point as can be seen in Figure 6.4. This is where we introduce a constraint, which relates to Kirchhoff's first law in electrical circuits. However, our implementation of this equation uses energy $E_{i,t}$ of a timestep.

$$\sum_{i=0}^{N} E_{i,t} = 0$$

where:

- N is the number of interconnected nodes in the many-to-many connection.
- $E_{i,t}$ is the energy of node i at time step t.

Listing 6.12 shows the implementation of the *ConnectingNode* class. This logical abstraction is used for connecting multiple nodes within the modelling graph, which allows us to represent many-to-many relationships. When nodes are connected via a *ConnectingNode*, a constraint is introduced to enforce Kirchhoff's first law (lines 10-14). In this implementation, this is achieved by ensuring that the power that flows into and out of the *ConnectingNode* at each timestep is zero.

```
1 class ConnectingNode(Node):
2     self.connected_nodes = []
3
4     def connect(self, node):
5         self.connected_nodes.append(node)
6
```

```
7       def connect_nodes(self, nodes):
8           self.connected_nodes.extend(nodes)
9
10      def constraints(self, t):
11          return [
12              cp.sum(
13                  [node.powerflow(t) for node in self.connected_nodes]
14              ) == 0]
```

Listing 6.12: ConnectingNode implementation

In Listing 6.13 we show an example usage of the *ConnectingNode* for the running example. After creating all relevant domain nodes and the connecting node (lines 1–8), the connections between them are done via the *connect_nodes* method (line 11). Alternatively, we can use the metering point as a connection node, which can be done by using the *connect_to* method that is available to any node (line 13). Behind the scenes, the same *ConnectingNode* gets created as before, so the choice of which approach to use is up to the user's preference. By modelling these connections, the constraint for ensuring that the household's load is met, is automatically introduced due to Kirchhoff's first law (as seen in Listing 6.12). If the sum of all powerflows from all nodes is zero, that means that there is enough production (or drawn from the grid) to cover the household's consumption.

```
1  # Create nodes
2  battery = Battery(...)
3  pv = SolarPanel(...)
4  household = Consumer(...)
5  metering_point = MeteringPoint(...)
6
7  # Create a connecting node
8  balance = ConnectingNode(...)
9
10 # Connect nodes to the balance node (ConnectingNode)
11 balance.connect_nodes([battery, pv, household, metering_point])
12
13 metering_point.connect_to([battery, pv, household])
```

Listing 6.13: Example usage of ConnectingNode

6.5.5 Domain-Specific Extensions

A key feature of this library is its extensibility through domain-specific subclasses. That is done by extending the base classes of *Node* and *GraphProblemClass*, therefore allowing support for a wide variety of real-world mathematical optimisation scenarios. In this iteration of this thesis, the focus is on the energy sector, therefore we have implemented built-in classes that the user can utilise to model optimisation problems within the energy industry.

Electricity market domain

We aligned the naming for the domain-specific classes according to the Harmonised Electricity Market Role Model (HEMRM)[37]. This can be seen on the class diagram in Figure 6.6. From HEMRM, we use the following two models:

- *Resource*: This represents consumption assets, production assets and energy storage assets.
- *Metering Points*: This represents the connection to the grid and where the produced and consumed electricity is measured.

Divergence from HEMRM: We made a small change in the class diagram in HEMRM, as it can be seen in Figure 6.5. The *Metering Point* and *Resource* are associated, but in our class diagram, the *Metering Point* inherits from the *Resource*.

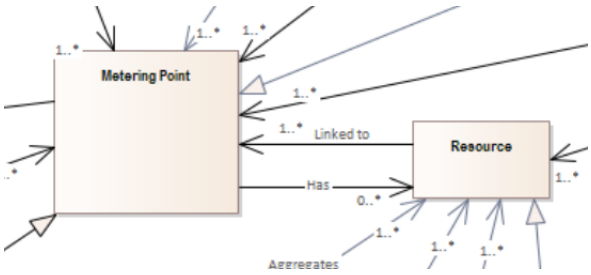

Figure 6.5: The connection between metering point and resource in HEMRM. The picture is taken from HEMRM. [37]

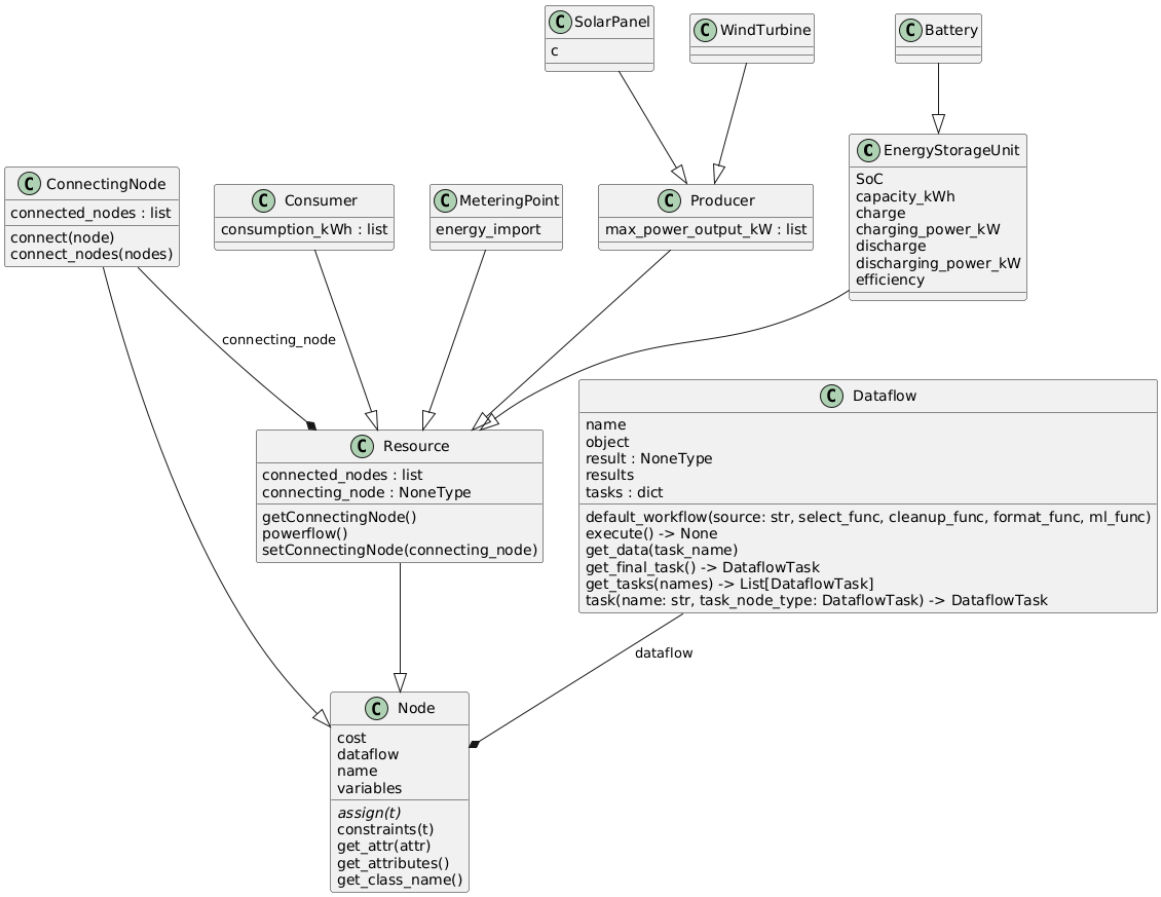

Figure 6.6: Class Diagram from the Node and Energy Domain classes. The diagram was generated with PlantUML. [38]

Resources

All of these classes inherit from a common *Resource* class and represent the following distinct components, which can be specialised to represent those assets' parameters. These classes can be seen on the class diagram in Figure 6.6.

- *Energy Storage Unit* is a class capable of storing electricity. It has an internal state, which is the amount of electricity stored. The class captures operational characteristics, like efficiency, maximum charge, and discharge power. This class can also be extended to include characteristics that are not in the base library, like battery degradation.
- *Producer* is a generic class for all production units, it captures operational characteristics like production schedules and cost of production per electricity unit, it can be extended to include selectable production schedules by the optimiser.

- *Consumer* is a generic class representing consumers with consumption schedules.

Each of the classes above inherits from the base *Resource* class. For example, Listing 6.14 shows a part of the implementation of the *Consumer* class. Due to inheritance, the *powerflow* method needs to be defined (lines 6-7). Furthermore, the consumption schedule of the consumer can either be assigned during construction of a *Consumer* object, or once all associated data flows are executed (*assign* method).

In the running example, we model the household as a *Consumer*. We are only interested in the consumption schedule, therefore, the *powerflow* property of *Consumer* is sufficient for our problem, as it returns the consumption at each time interval as a negative amount, which we then try to offset with production.

```
1  class Consumer(Resource):
2      def __init__(self, name, consumption_kWh=[]):
3          super().__init__(name)
4          self.consumption_kWh = consumption_kWh
5
6      def powerflow(self, t):
7          return -self.consumption_kWh[t]
8
9      def assign(self, t):
10         self.consumption_kWh =
11         self.dataflow.results["gen_consumption"]["consumption_kWh"][:t]
12             .values
```

Listing 6.14: Partial implementation of *Consumer* class

Producer specialization

- *Solar Panel* is a class that represents the solar panels, which is a specialisation of the Producer class, with a controllable output set by the optimiser, and a maximum production schedule based on the irradiation forecasts.
- *Wind Turbine* is a class that represents the wind turbine, which is a specialisation of the Producer class, it has a production schedule based on the wind forecasts.

Energy Storage Unit specialisation

The *Battery* class is a concrete specialisation of the *EnergyStorageUnit* base class. As shown in Listing 6.15, it models the energy storage behaviour of a battery over time. The charge and discharge schedules are determined by the optimiser, while operational constraints

such as power and capacity limits are enforced by the class.

```
class Battery(EnergyStorageUnit):
(...)
    def constraints(self, t):
    constraints = [
        self.SoC[t] <= self.capacity_kWh,
        self.charge[t] <= self.charging_power_kW,
        self.discharge[t] <= self.discharging_power_kW,
    ]
    if t == 0:
        constraints.append(
            self.SoC[t] == self.efficiency *
                self.charge[t] - (1 / self.efficiency) * self.discharge[t])
    else:
        constraints.append(
            self.SoC[t] == self.SoC[t-1] + self.efficiency * self.charge[t] -
                (1 / self.efficiency) * self.discharge[t])
    return constraints

def powerflow(self, t):
    return self.discharge[t] - self.charge[t]
```

Listing 6.15: Implementing the Battery class

The *BatteryWithDegradation* class is an extension to the *Battery* class by introducing a cost term that models battery degradation. The cost accounts for the depreciation of the battery's value as it ages with charge and discharge cycles. The degradation model is adopted from [39]. This class is not used in the examples and experiments. It demonstrates how the *Battery* implementation can be extended with additional domain-specific logic.

Listing 6.16 shows how the *BatteryWithDegradation* class introduces a cost function based on a degradation model.

```
class BatteryWithDegradation(Battery):
    def __init__(..., efficiency=0.9, D50=1.0, beta=0.693):
        super().__init__(...)
        self.D50 = D50
        self.beta = beta

    def cost(self):
```

```
8           return self.charge * self.D50 *
9               np.exp(self.beta * ((self.SOC - 50) / 50))
```

Listing 6.16: Extending the Battery class with degradation.[39]

Chapter 7

Experiments

Throughout this section, we will be comparing the performance, productivity and scalability aspects of our tool. For the performance comparisons, we will be looking to compare problem implementations in CVXPY, Pyomo, GBOML with our GPO-D solution. The scalability experiments will look at the limitations of our GPO-D library. Lastly, we measure the productivity of GPO-D compared with CVXPY, Pyomo and GBOML through the number of characters and effective lines of code used, and our experiences developing with it. The figures in this chapter were generated with the *matplotlib* library.

7.1 Comparisons with Other Python Implementations

This section focuses on comparing our GPO-D implementation of the microgrid example described in chapter 5 with other Python optimisation libraries such as CVXPY, Pyomo or GBOML. The goal here is not to outperform these libraries, but rather to keep the performance levels similar, especially with CVXPY, which is also the library employed in GPO-D. Furthermore, data processing and ML workflows stay the same for all the implementations, and only optimisation problem solutions differ from library to library.

7.1.1 Performance and Memory Usage

Setup

In these experiments, we will be benchmarking execution time and memory usage of the microgrid example implemented in GPO-D, CVXPY, Pyomo and GBOML. Specifically, the microgrid setup for the experiments is composed of components listed in Table 7.1. The actual microgrid implementations for each of the libraries is in appendix section A.2.1 for GPO-D, appendix section A.2.2 for CVXPY, appendix section A.2.3 for Pyomo, and appendix section A.2.4 for GBOML solution. For more insights, the time horizon and number of homes with solar panels will be scaled as follows:

Microgrid Component	**Count**	Parameters
Time horizon T	24	in hours
Homes	50	consumption factor 1
Schools	1	consumption factor 100
Solar panels (for homes)	50	5kW rated production capacity
Solar panels (for schools)	1	25kW rated production capacity
Wind turbines	1	1000 kW rated production capacity cut in speed of 3.5 m/s cut out speed of 25 m/s rated speed of 14 m/s
Batteries	3	500 kWh storage 0.9 efficiency 500 kW of charge and discharge power initial state of charge is 0 kWh

Table 7.1: Default microgrid setup for the experiments

- time horizon scaling in figures 7.1 and 7.3
  - 24x1 (1 day)
  - 24x2 (2 days)
  - 24x3 (3 days)
  - 24x5 (5 days)
- number of homes and solar panels scaling in figures 7.2 and 7.4
  - 50 homes and 50 (home) solar panels
  - 100 homes and 100 (home) solar panels
  - 200 homes and 200 (home) solar panels
  - 500 homes and 500 (home) solar panels
  - 1000 homes and 1000 (home) solar panels

The main idea of this scaling is to increase the overall number of tasks (such as solar irradiation retrieval, cleanup, and calculation of max. rated production capacity for solar panels) in data processing workflow, overall number of tasks (such as predicting consumption for homes with pre-trained model) in ML workflow as well as increasing the complexity of optimization problems through growing number of constraints and variables that come with this scaling.

As for the data setup, the same prediction model is used to predict consumption over T for homes and school with results multiplied by factors defined in table 7.1 respectively

Tool	Version
Python	3.10.8
cvxpy	1.6.0
gboml	0.1.10
pyomo	6.9.2

Table 7.2: Versions of tools used for the experiments

by 1 for homes and by 100 for school. For wind turbine and solar panels for both homes and school, the same location (Aalborg location) is used to retrieve real-time wind speed and solar irradiation over T period of time. Lastly, all batteries start with the initial state of charge at 0 kWh.

These experiments are run on a laptop with Windows 11, Intel Core i5 CPU, and 16.0 GB of RAM. They are executed in Python version 3.10.8 and its related tools are listed in table 7.2.

Results

Looking at the first set of execution time measurements wtih time horizon scaling in figure 7.1 for original 50 homes and 50 (home) solar panels, it can be noticed that overall with scaling number of days there is a small linear increased of 1-2 seconds per added day for our GPO-D and CVXPY solutions whereas Pyomo and GBOML results remains all under 7 seconds of execution, not affected by time horizon scaling. Moreover, it is only the optimisation part being impacted by the scaling, as the dataflow execution times are very similar (around 5.8 seconds) across the whole x-axis, with GBOML slightly slower (around 6.4 seconds mark). The reason why dataflow execution doesn't change much is because the number of dataflow tasks remains the same, it is just the size of the datasets (from 24 to 24x5=120 rows of data for components in the microgrid) that varies, which for our small datasets is negligible. On the other hand, the optimisation problem complexity rises by the added day, for example, for 3 days there are 3 times more variables and constraints, or for 5 days there are 5 times more variables and constraints, all compared to 1 day. The actual number of variables and constraints is listed in the appendix section A.3. To sum up, GPO-D seems to perform the worst out of the 4 solutions across the board, being only very slightly behind the CVXPY solution. The fastest solution is Pyomo, which takes around 6 seconds to execute across all time horizons, and the second in place is GBOML, which is often around 0.7-1 second slower compared to Pyomo, mainly due to slightly slower dataflow execution compared to the average.

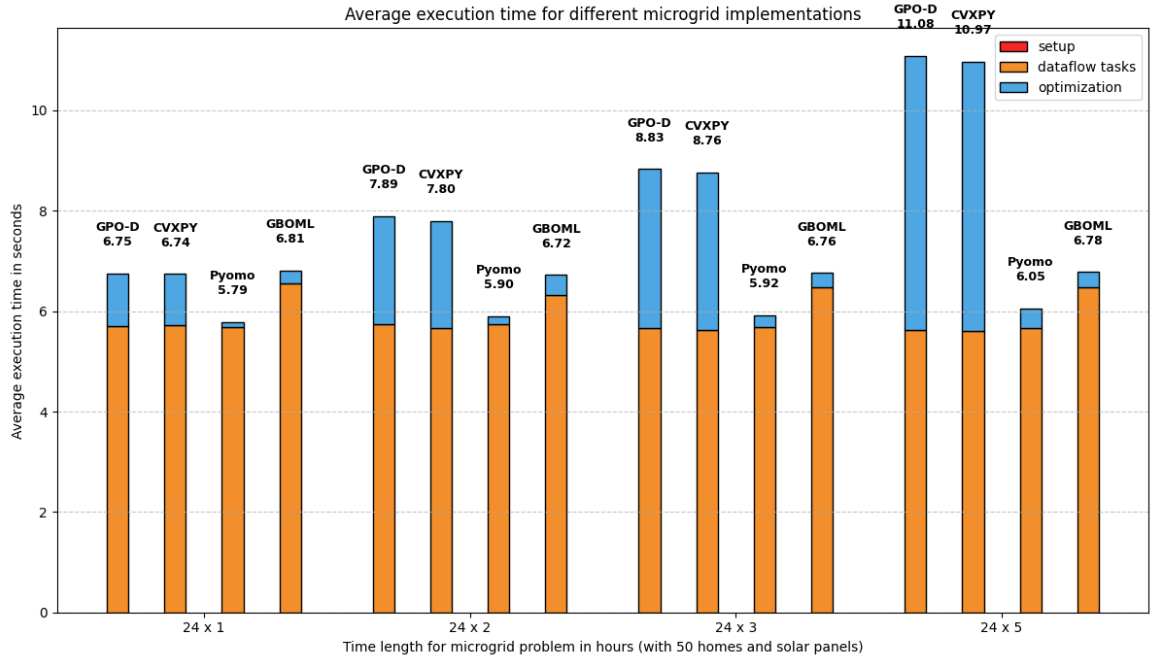

Figure 7.1: Execution time comparison between GPO-D, CVXPY, Pyomo and GBOML with scaling time horizon

In the second set of measurements displayed in the figure 7.2, the number of homes and the number of solar panels for homes are being scaled. Generally, a linear trend can be seen in the results where the total execution time is for 100, 200, 500 and 1000 homes each with 1 solar panel is 2x, 4x, 10x, and 20x slower compared to the execution times of the first group of 50 homes and their solar panels. Once again, as observed in figure 7.1, GPO-D and CVXPY optimization executions are keeping up very closely across the groups with GPO-D being slightly slower whereas Pyomo and GBOML executes optimization phase considerably faster which often takes under 1 second and not impacted by the overall increase in number of constraints and variables (where with 1000 homes and 1000 solar panels there is over 24000 variables and 48000 constraints as seen in appendix section A.3). Note that GBOML fails with 500 and 1000 of homes and solar panels which it is assumed to be due to how the optimization problem is constructed, specifically with solar panel nodes being almost copy pasted in order to be able to create a decision variable associated with a unique dataset (not done in our microgrids solutions but each solar panel can have different coordinates which retrieve different values for solar irradiation and total energy production varies for each solar panel) for the time horizon T for each of the solar panels nodes. There is possibly a solution to fix these issues, however, due to a lack of documentation and similar examples (examples where there is a great number of decision variables, each dependent on a unique dataset), the actual issue or solution has not been found. The last thing to point out is that dataflow takes up more than 80% of the execution

time for all of the solutions across the scale. To conclude, GPO-D results are very similar to those of CVXPY across the board, and GBOML comes in second after Pyomo, which can execute the dataflow tasks a couple of seconds faster.

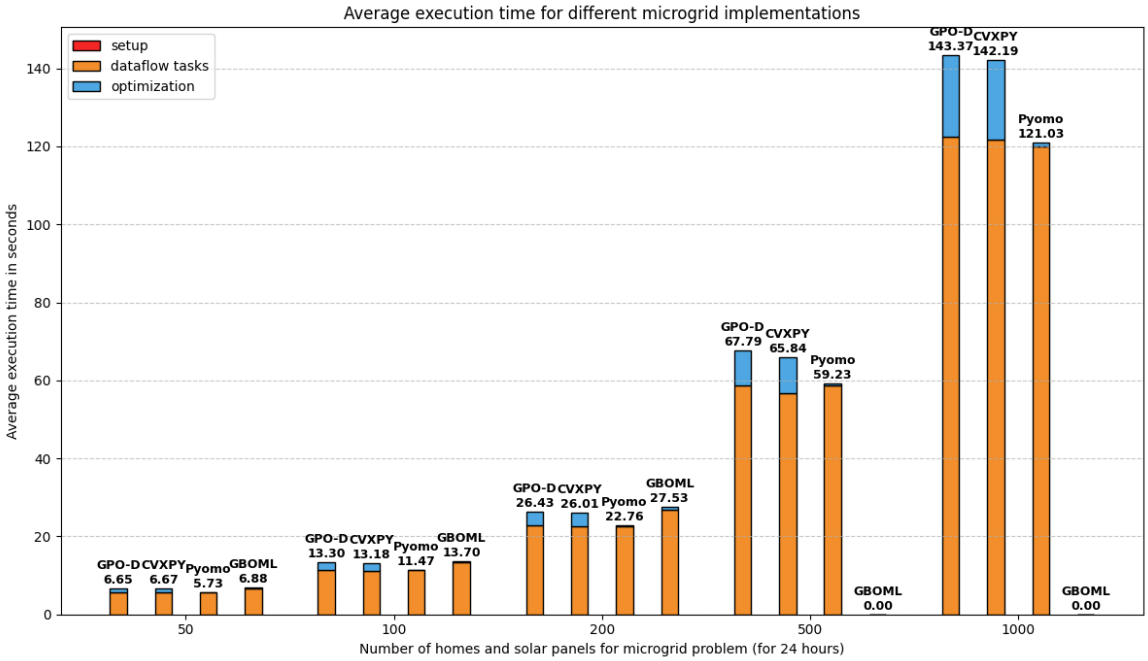

Figure 7.2: Execution time comparison between GPO-D, CVXPY, Pyomo and GBOML with scaling number of homes and (home) solar panels

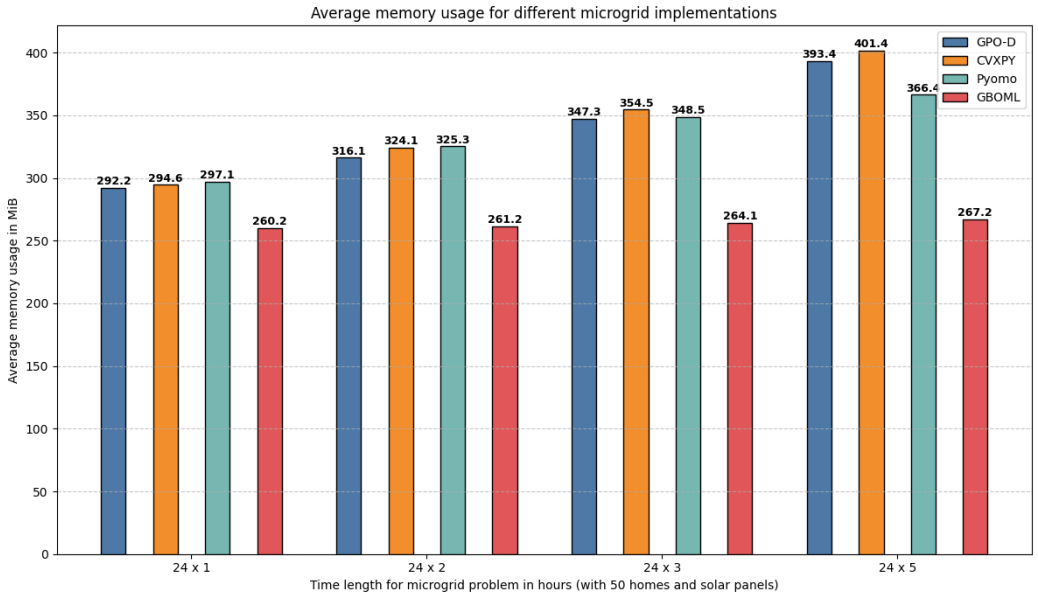

Figure 7.3: Memory usage comparison between GPO-D, CVXPY, Pyomo and GBOML with scaling time horizon

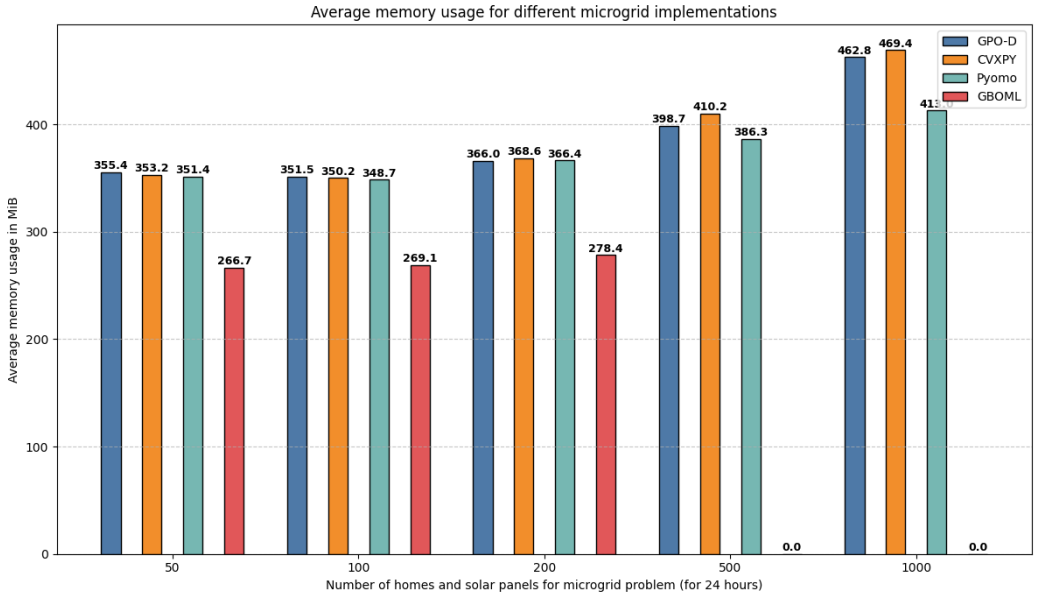

Figure 7.4: Memory usage comparison between GPO-D, CVXPY, Pyomo and GBOML with scaling number of homes and (home) solar panels

Memory usage experiments in figures 7.3 and 7.4 are also scaled across the time horizon and the number of homes and solar panels. Overall, across all the microgrid solutions,

except for GBOML, similar levels of memory usage with a slight increase of around 20 MiB across the time horizon scale in figure 7.3 for each of the respective solutions. GBOML, however, uses the least memory (around 260 MiB to 280 MiB) during its execution and not really affected by the scaling of time horizon or number of homes and solar panels mainly due to storing the data inside the csv files (the data is retrieved when nodes and edges are imported into the GBOML model) instead of keeping them in memory in a dataframe or a numpy array throughout the execution of the solution.

7.1.2 Productivity Experiments

For the quantitative analysis, we measure effective lines of code (eLoC) and number of characters (NoC). To compare against libraries, it needs to be pointed out that those two metrics are not an effective way to measure developer productivity, since they do not capture library learning complexity and usability complexity. These are both subjective measurements and are hard to quantify in an objective manner.

Setup

For measuring eLoC and NoC, we use the same implementations as we do for performance and memory usage, and we consider only the base scenario as described in Table 7.1. We strip away any comments, unused variables, and imports from all implementation files in order to get the most accurate results. Furthermore, for GPO-D, we take two measurements - one excluding domain classes (e.g., Battery class), and another including the domain classes. That is done because on one occasion, the built-in domain classes can be sufficient for building the solution, while in other situations, one may need to extend the library with additional domain classes. Similarly, for GBOML, we measure with and without the input text file that contains all the components and parameters for the problem. Since the input text file uses a different language than Python, we measure LoC instead of eLoC.

For measuring eLoC, we use *cloc* [40], which is a universal tool for counting lines of source code, comment lines, and blank lines [40]. For counting NoC, we simply use a shell command that counts the number of characters excluding whitespaces.

```
tr -d '[:space:]' < /workspaces/gpod_microgrid_solution.py | wc -m
```

Listing 7.1: Shell command for counting NoC

Results

The results of the eLoC and NoC measurements can be seen in Figure 7.5 and Figure 7.6 respectively, and the concrete values can be found in Appendix section A.3.2. As observed, the base implementations with GPO-D and GBOML, without including domain

classes or input text files, are smaller than CVXPY and Pyomo in both eLoC and NoC. A key role in this difference are the abstractions provided by GPO-D and GBOML. However, if we take into consideration the domain classes or text files needed by GBOML, then the difference turns into CVXPY and Pyomo's favour. For GPO-D, this would only be the case if additional domain classes need to be added, or an extension to another domain needs to be implemented. For GBOML, a problem cannot be formulated or solved without creating the input text file. The large number of LoC and NoC that can be seen in the figures is due to the fact that GBOML doesn't support loops as part of the language and grammar it uses. Therefore, this requires the development of a program that can write such files by essentially repeating the definition of a node n number of times. To conclude, solutions implemented with GPO-D tend to be the most concise, given that all required domain classes are already provided by GPO-D. When that is not the case, GPO-D solutions can take more time to implement and end up larger in code size than their counterparts.

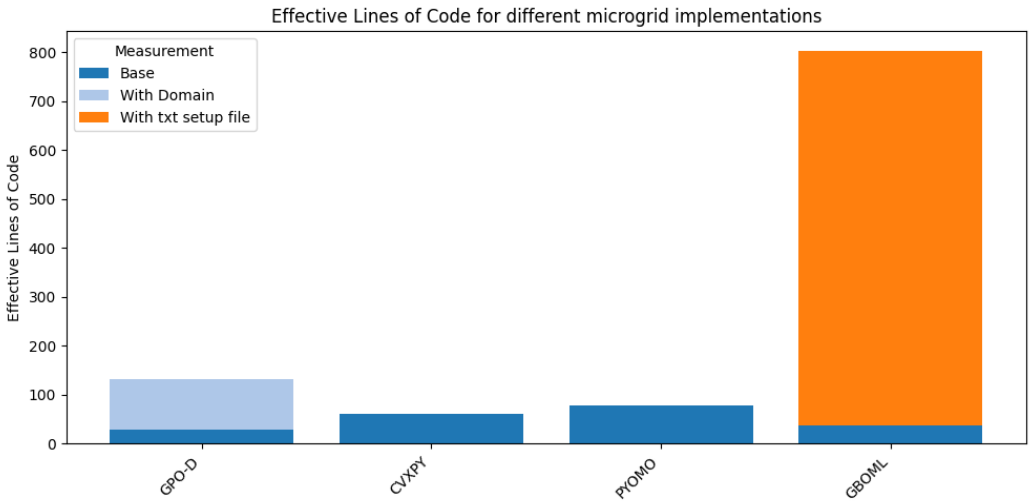

Figure 7.5: Effective lines of code comparison between GPO-D, CVXPY, Pyomo and GBOML

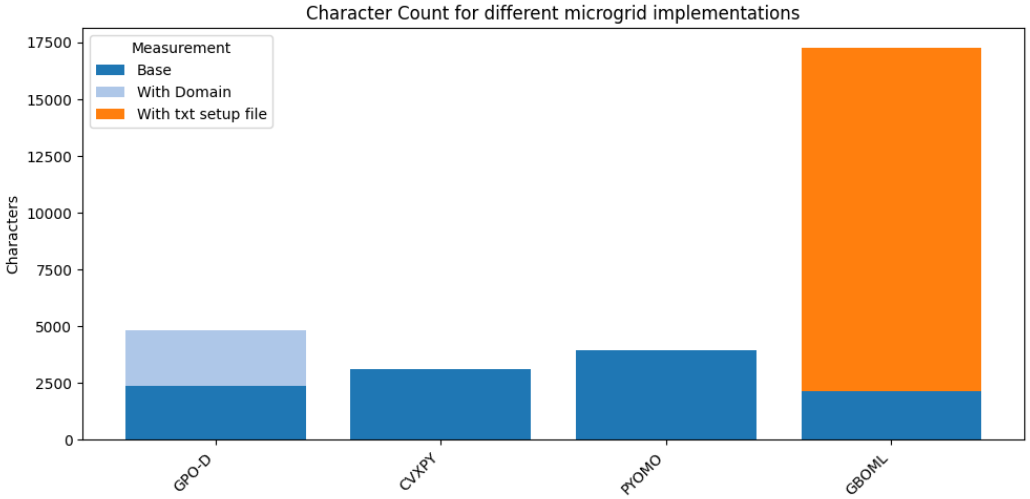

Figure 7.6: Number of characters comparison between GPO-D, CVXPY, Pyomo and GBOML

7.2 Scalability Experiments

In these experiments, we will be scaling up the problem size to see how execution time changes.

Setup

We will also be looking at the parallel execution method for the *Dataflows*. The microgrid setup for the experiments is composed of components listed in Table 7.1.

The experiments were run on a laptop with Windows 11, AMD Ryzen 5 CPU and 24 GB of ram and a 1660 Ti Max-Q Nvidia GPU. They are executed in Python version 3.10.8 with a custom build of CVXPY from GitHub, which included the *cuOpt* solver in the cvxpy library[1].

- time horizon scaling
  - 1 day
  - 2 days
  - 4 days
  - 7 days
- number of homes and solar panels scaling
  - 10

[1] `https://github.com/cvxpy/cvxpy/pull/2820`

  - 50
  - 100
  - 500

- changing the length modifier to simulate 15 minute timesteps. This was previously introduced in subsection 5.1.3.
  - 1
  - 4

7.2.1 Optimization time

We looked at the optimisation time required to solve a problem with regard to the problem size. This was calculated by adding up all generated Variables and Constraints, which were returned from cvxpy. We used the *CBC*[41] and *cuOpt* solvers. cuOpt is an open-source GPU-accelerated optimizer[42]. The results can be seen in Figure 7.7.

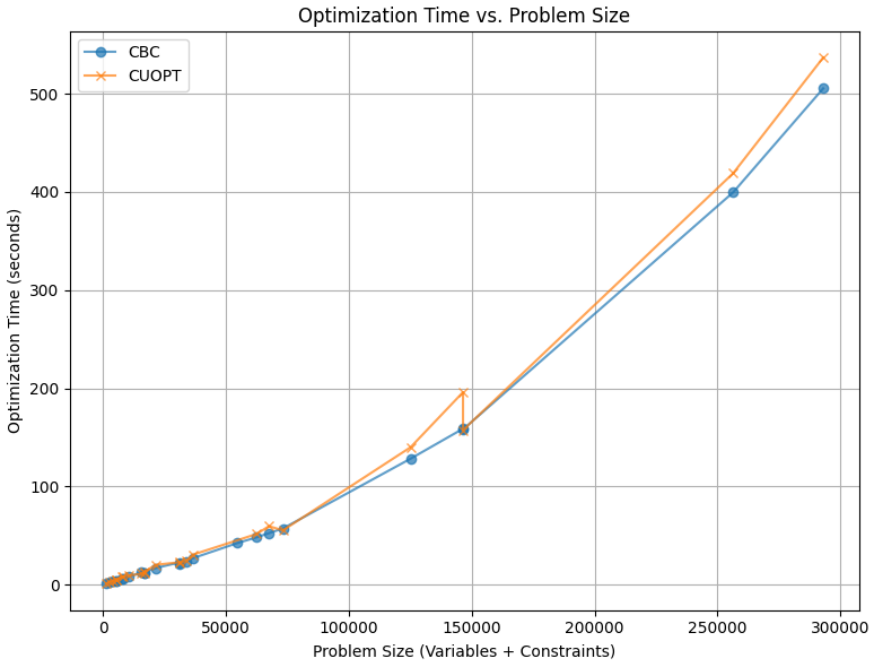

Figure 7.7: Optimization time for cuOpt and CBC[41] solvers

Results

Both solvers present similar scaling, however *CBC* offers a bit better performance, while *cuOpt* has some unpredictability.

7.2.2 Parallel dataflow processing

As an attempt to improve the overall performance of *dataflows*, we implemented a *ParallelDataflowManager* with the same interface as the *DataflowManager*, but with one key difference of executing *Dataflows* in a *ThreadPoolExecutor* from the *concurrent.futures* Python library. This allows us to introduce some parallelism to the *Dataflow* processing.

We compared the dataflow processing times with regard to the number of households in parallel and serial processing modes. The results of this comparison can be seen in Figure 7.8. We only compared against the different number of households and connected solar panels, since that decides the number of dataflows that need to be processed.

Results

The parallel implementation did not improve the *Dataflow* processing times. It seems that the overhead of creating and scheduling threads is more significant than the time gained by the concurrent processing.

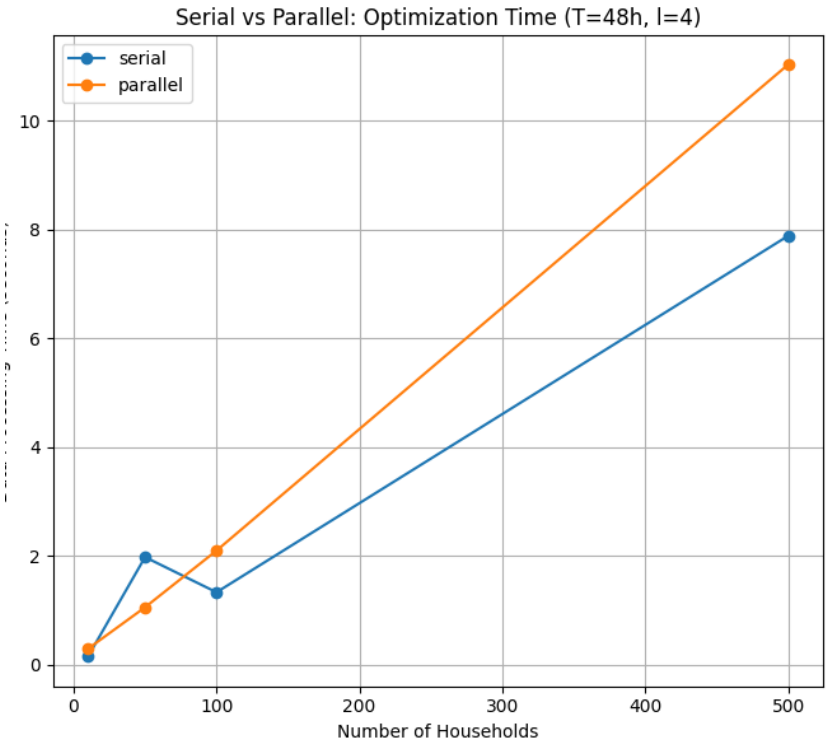

Figure 7.8: Serial vs Parallel comparisons

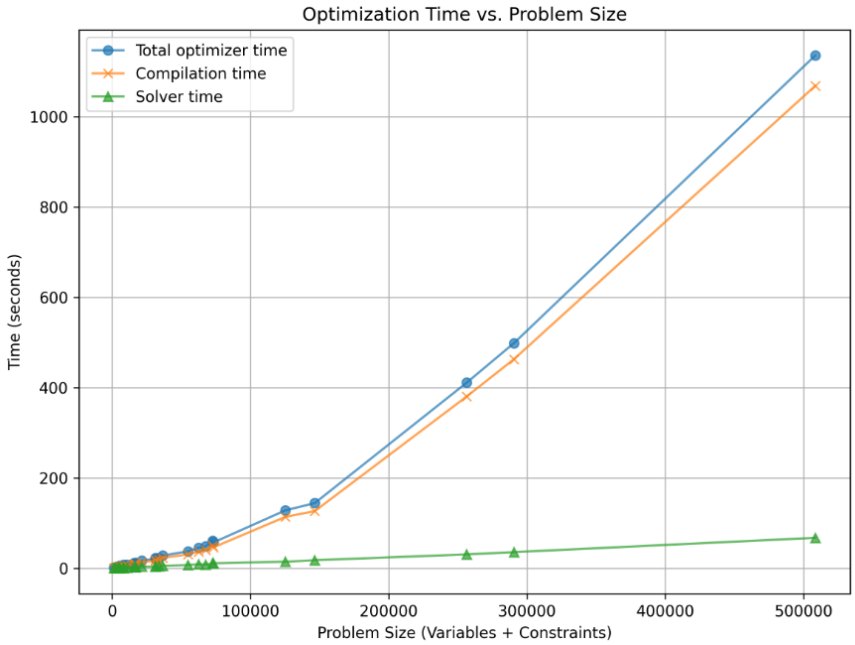

Figure 7.9: Compilation Time measurements

Difficulties in testing the parallel processing

These parallel tests were run with a function to mock the consumption data instead of *pycaret*. This was due to the incompatibility of *pycaret* with concurrent execution. The tests with that library included produced indeterministic errors. These errors are likely due to an internal global state, which can be conflicted if run concurrently.

7.2.3 Measuring CVXPY compilation time

CVXPY performs several internal steps before handing the problem over to the solver, including symbolic expression parsing, disciplined convex programming compliance checks, which ensure that the problems are convex, and canonicalization to a standard solver-compatible form[43, 44]. These steps can introduce a significant overhead. To test this assumption, we measured the compilation times of CVXPY and compared them with the total time taken to solve the optimisation problem. These measurements can be seen in Figure 7.9. After that, we also generated a figure for the ratio of compilation time to total solver time, which can be seen in Figure 7.10. This experiment was performed with the CBC solver.

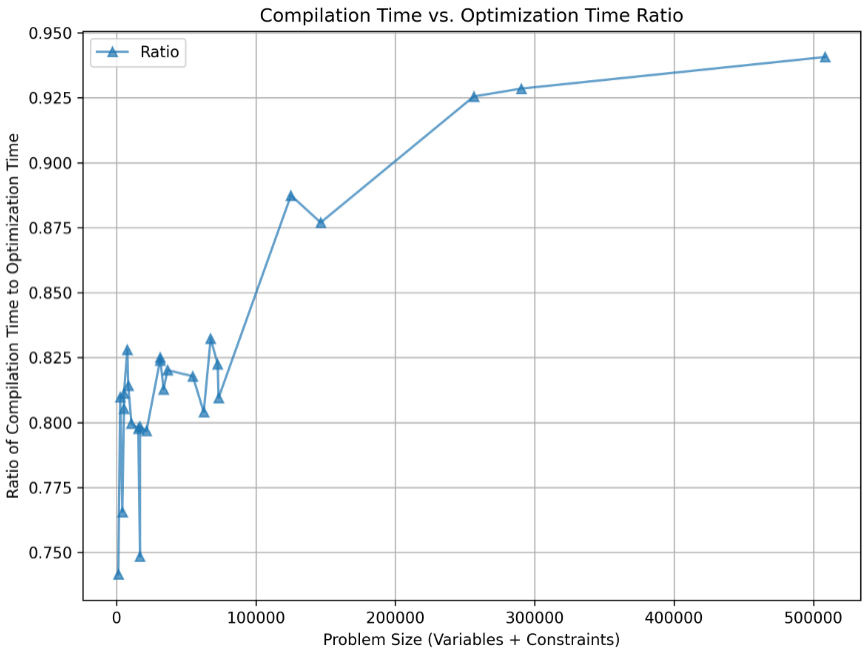

Figure 7.10: Compilation Time ratio

Results

In Figure 7.9 we scaled the problem size by increasing the number of timesteps and the number of homes (and their solar panels) and looked at the time taken for *cvxpy* to compile the problem and the total time to solve it with compilation included. From this, we could subtract the time taken by the solver. The compilation takes significantly longer than to actually solve the problem. To represent the ratio we plotted it in Figure 7.10. From this figure, we can see that in the best case scenario, the compilation takes a bit less than 75% of the time, while in the worst case scenario with bigger problem sizes, it takes up 94% of the total solve time.

7.3 Discussion

It has to be pointed out that the amount of code does not map one-to-one to developer productivity, because it fails to measure the difficulty of using a given tool or library. To confirm developer productivity improvements, we would need to do user surveys, which we do not have the time or resources to conduct.

7.3.1 Segmentation of workflows

Although we did not measure developer productivity through surveys. During our development of the examples, we did notice the separation of workflows between the im-

plementation of the predictions and the implementation of the microgrid optimisation model, and we assume this would make these workflows easier to execute. We recognise that these are still assumptions and a proper user study would need to be done in order to support this claim.

7.3.2 CVXPY overhead

Throughout the experiments, we observed that *cvxpy* introduces a noticeable overhead, which we quantified in Subsection Measuring CVXPY compilation time 7.2.3. As the problem size increases, this overhead becomes more significant and limits the scalability potential of GPO-D. Given this limitation, we may need to reconsider the use of *cvxpy* as our main optimisation library, especially for large-scale problems.

Chapter 8

Conclusion

This thesis set out a goal of developing a library for PSA named GPO-D. The main problems solved are improving user-friendliness and developer productivity, while retaining the support for the full PSA workflow and keeping up with performance in terms of execution speed when compared to CVXPY, Pyomo and GBOML. A small running example together with its solution in GPO-D is introduced in Chapter 1. In Chapter 2, similar works in the field of Prescriptive Analytics, such as SolveDB+ and GBOML, are presented through our small running example. In Chapter 3, an analysis of the PSA workflow, what it consists of and how it is expanded for our purpose is showcased together with research gaps that exist in the current tools and technologies. Later on in the chapter, a set of requirements for GPO-D is listed in order to bridge the aforementioned gaps. Lastly, the main focus for the GPO-D is on problems from the energy sector, due to our experience in it. In Chapter 4, 3 research questions are presented to guide us in the development of the GPO-D Python library that supports the full PSA workflow with the focus on developer productivity, user-friendliness, simplicity and similar performance compared to CVXPY, Pyomo and GBOML. Based on that, here are the research questions and their answers:

RQ1: What abstraction(s) are needed in our tool to make the development of a full PSA solution simpler and more oriented towards developer productivity?

Answer for RQ1: Graph modelling abstraction, which is described in section 6.5, is implemented to make the definition of optimisation problems simpler, for example, collecting decision variables, constraints and constructing an objective function through the custom *GraphProblemClass*, thus abstracting away some of the implementation logic that otherwise is necessary to be defined by the developer in libraries like CVXPY or Pyomo. Moreover, built-in pre-defined components of the energy sector, such as *Consumer*, *Producer*, or *Battery*, increase the developer productivity because of how these classes are reused in other problems from the energy sector, such as battery arbitrage, where the *Battery* object is used. The experiments conducted in section 7.1.2 confirm and showcase the potential of using

less code while achieving the same results, for instance, implementing the microgrid solution, which is presented in the chapter 5, with already built-in microgrid domain classes in GPO-D takes less amount of code compared to CVXPY, Pyomo or GBOML implementations.

The second abstraction presented is the Dataflow, also described in section 6.4. Dataflows that are associated with domain nodes, such as *SolarPanel* or *WindTurbine* used in the microgrid solution, provide the ability to add and execute a number of tasks from data processing or ML workflows. These dataflows with their tasks, which are executed through the *GraphProblemClass*, further simplify the overall PSA workflow as the data necessary for the optimisation problem get to the right domain classes.

RQ2: How do we make our abstraction(s) extensible in a way that developers can implement different PSA optimisation problems from the energy sector?

Answer for RQ2: The graph modelling abstraction further includes core abstractions described in the subsection 6.5.2 and objective function constructor described in the subsection 6.5.3. The core abstractions, which are the *Node* and *GraphProblemClass*, provide a generic way to extend the energy domain through the inheritance of the base *Node* class and build the overall optimisation problem with the *GraphProblemClass*. To get one level deeper, the objective function constructor specifically helps build an objective function that is decomposed in the domain nodes that are associated with the *GraphProblemclass*. To sum up, the abstractions for optimisation problems are implemented in an object-oriented manner, using components as classes, and to make extensibility easier. HEMRM [37] is employed as a baseline for the naming and making the re-usability more standardised.

RQ3: How does a problem example solution implemented in GPO-D compare to the implementations in CVXPY, Pyomo and GBOML in terms of performance?

Answer for RQ3: In this case, microgrid problem example solution (described in the chapter 5) is implemented in GPO-D (full code in appendix section A.2.1), CVXPY (full code in appendix section A.2.2), Pyomo (full code in appendix section A.2.3), and GBOML (full code in appendix sections A.2.4). In the experiments subsection A.3, a comparison in terms of execution speed and memory usage performance is conducted between GPO-D, CVXPY, Pyomo and GBOML microgrid implementations, with execution speed being more interesting for us. The results show that GPO-D closely keeps up with the CVXPY in terms of execution speed, however, it is Pyomo that performs the best. On that note, as part of the future work, it may be worth considering using Pyomo as part of our library instead of CVXPY or possibly exploring other optimisation libraries such as SciPy to see if they are a better fit.

To conclude, a functional prototype of the GPO-D Python library has been developed with graph modelling and dataflow abstractions, the CVXPY library employed for optimisation problem definitions, and PyCaret used for predictions (currently having only one method for predicting the consumption). Microgrid problem example has been successfully implemented in GPO-D and in other implementations, namely CVXPY, Pyomo and GBOML, and all of them return the same optimisation values based on the same datasets provided for them.

8.1 Future Work

Improving Data Handling between *Dataflow Tasks*

In the current implementation of GPO-D, data between *tasks* in a *dataflow* is passed as a dictionary of dataframes. Upon completion of the execution of a *task*, the resulting dataframe is merged into the dictionary and then forwarded to the next task in the dependency chain. A major drawback is that the dictionary will become increasingly larger in size as the number of *tasks* grows. That can lead to memory and further performance issues. A viable option is to look into utilising DuckDB for solving that. The dataframe from a *task* would be inserted into DuckDB, and when another *task* starts its execution, the required dataframes would be queried from the database.

DuckDB is an SQL OLAP database management system which is designed for fast analytical query execution [45]. It can efficiently handle querying large datasets from disk or memory, and it can integrate seamlessly with *pandas*, thus allowing for hybrid SQL-Python workflows [45, 46].

Storing and Loading Optimisation Problem Graph Models

As seen in the code example for GBOML in Chapter 2, the entire optimisation problem, which is modelled as a graph, is written in a text file that can be loaded into the *GbomlGraph* class on demand. Similarly, in GPO-D, we can implement this feature by mapping the *GraphProblemClass* and all related *Nodes* into a JSON format that can then be stored in a DBMS like DuckDB. This would greatly improve developer productivity, as it would allow constructing large optimisation problems with only a few lines of code needed for loading them from a database.

Graphical User Interface

Currently, GPO-D is designed for programmatic use through Python, primarily targeting data scientists who are comfortable with code. While this provides full control and flexibility over creating PSA workflows with GPO-D, it limits accessibility for non-technical users. Therefore, the development of a graphical user interface (GUI) would be a great feature

to have in GPO-D. It would significantly improve the usability by allowing users to define optimisation problems, configure dataflows and forecasting models, view visualisations of dataframes, and set optimisation objectives through an intuitive interface.

8.1.1 Domain Extensibility

While initially designed specifically for energy-related applications, the generic base classes make extensions and adoptions to other prescriptive analytics domains, such as logistics, manufacturing, inventory management, etc., suitable. For instance, the generic base classes can easily be extended to support solving the Newsvendor [47] problem.

Newsvendor Problem

The *Newsvendor* class, as shown in Figure 8.1, models uncertainty in demand and ordering decisions, which is a classic problem in inventory management [47]. The class handles stochastic inventory problems by modelling demand scenarios, ordering costs, and potential shortages of products, as well as capturing cost components such as holding and shortage penalties. Additionally, it accounts for product expiration dates, which is relevant in contexts involving perishable goods, as that leads to more realistic and efficient inventory decisions.

This shows the flexibility of the graph-based approach that can be generalised beyond the energy sector.

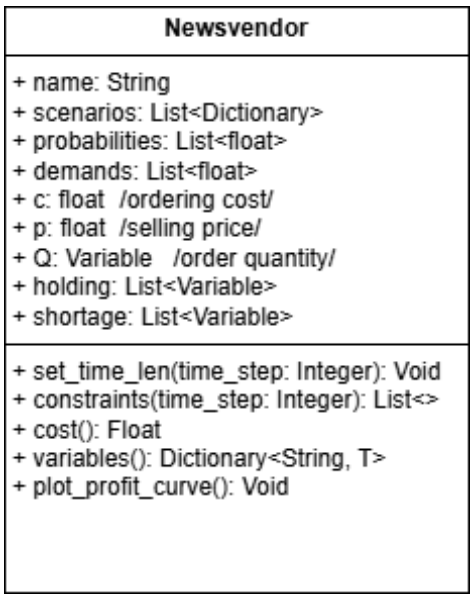

Figure 8.1: Class diagram of the specialised class for the Newsvendor problem

Bibliography

[1] Christopher Wissuchek and Patrick Zschech. "Prescriptive analytics systems revised: a systematic literature review from an information systems perspective". In: *Information Systems and e-Business Management* (2024). DOI: 10.1007/s10257-024-00688-w.

[2] Katerina Lepenioti et al. "Prescriptive analytics: Literature review and research challenges". In: *International Journal of Information Management* 50 (2020), pp. 57–70. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2019.04.003. URL: https://www.sciencedirect.com/science/article/pii/S0268401218309873.

[3] Marvin Niederhaus et al. "Technical Readiness of Prescriptive Analytics Platforms: A Survey". In: *2024 35th Conference of Open Innovations Association (FRUCT)*. 2024, pp. 509–519. DOI: 10.23919/FRUCT61870.2024.10516367.

[4] Freepik. *Electric meter - Free technology icons*. [Online; accessed 2025-06-04]. Feb. 2019. URL: https://www.flaticon.com/free-icon/electric-meter_1548050.

[5] Iconjam. *Power grid - Free icons*. [Online; accessed 2025-06-04]. July 2024. URL: https://www.flaticon.com/free-icon/power-grid_17306079.

[6] Laurynas Šikšnys and Torben Bach Pedersen. "SolveDB: Integrating Optimization Problem Solvers Into SQL Databases". In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. SSDBM '16. Budapest, Hungary: Association for Computing Machinery, 2016. ISBN: 9781450342155. DOI: 10.1145/2949689.2949693. URL: https://doi.org/10.1145/2949689.2949693.

[7] Laurynas Siksnys et al. "SolveDB+: SQL-Based Prescriptive Analytics". In: *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*. Ed. by Yannis Velegrakis et al. OpenProceedings.org, 2021, pp. 133–144. DOI: 10.5441/002/EDBT.2021.13. URL: https://doi.org/10.5441/002/edbt.2021.13.

[8] Mathias Berger et al. "Remote renewable hubs for carbon-neutral synthetic fuel production". In: *Frontiers in Energy Research* 9 (2021), p. 671279. DOI: 10.3389/fenrg.2021.671279. URL: https://www.frontiersin.org/journals/energy-research/articles/10.3389/fenrg.2021.671279.

[9] Mathias Berger et al. “Graph-Based Optimization Modeling Language: A Tutorial”. In: (2021).

[10] Bardhyl Miftari et al. *GBOML Documentation*. https://gboml.readthedocs.io/en/stable/index.html. Accessed: 2025-05-05. 2025.

[11] The Apache Software Foundation. *Apache Airflow*. Accessed: 2025-05-08. 2025. URL: https://airflow.apache.org/.

[12] Ankit Chaurasia. *Airflow Survey 2024*. https://airflow.apache.org/blog/airflow-survey-2024/. Accessed: 2025-05-08. 2025.

[13] Jerin Yasmin et al. “An empirical study of developers’ challenges in implementing Workflows as Code: A case study on Apache Airflow”. In: *Journal of Systems and Software* 219 (2025), p. 112248.

[14] Davide Frazzetto et al. “Prescriptive analytics: a survey of emerging trends and technologies”. In: *The VLDB Journal* 28 (2019), pp. 575–595.

[15] Stefan Studer et al. “Towards CRISP-ML(Q): A Machine Learning Process Model with Quality Assurance Methodology”. In: *arXiv preprint arXiv:2003.05155* (2020). URL: https://arxiv.org/abs/2003.05155.

[16] Larysa Visengeriyeva et al. *CRISP-ML(Q): The ML Lifecycle Process*. Accessed: 2025-05-18. 2020. URL: https://ml-ops.org/content/crisp-ml.

[17] Vikash Mansinghka et al. *BayesDB: A probabilistic programming system for querying the probable implications of data*. 2015. arXiv: 1512.05006 [cs.AI]. URL: https://arxiv.org/abs/1512.05006.

[18] Bardhyl Miftari et al. “GBOML: Graph-Based Optimization Modeling Language”. In: *Journal of Open Source Software* 7.72 (2022), p. 4158. DOI: 10.21105/joss.04158. URL: https://doi.org/10.21105/joss.04158.

[19] *SQL vs. Python: A Comparative Analysis for Data | Airbyte — airbyte.com*. https://airbyte.com/blog/sql-vs-python-data-analysis. [Accessed 18-05-2025].

[20] Samuel Ivan, Simeon Plamenov Kolev, and Bence Schoblocher. *Exploration into Unified Tool for Prescriptive Analytics*. Unpublished report. AAU CS-IT 9th semester report. 2025.

[21] Sneha Gathani et al. *What-if Analysis for Business Professionals: Current Practices and Future Opportunities*. 2025. arXiv: 2212.13643 [cs.HC]. URL: https://arxiv.org/abs/2212.13643.

[22] U.S. Department of Energy, Grid Deployment Office. *Microgrid Overview*. Tech. rep. US Department of Energy, 2024. URL: https://www.energy.gov/sites/default/files/2024-02/46060_DOE_GDO_Microgrid_Overview_Fact_Sheet_RELEASE_508.pdf.

[23] Fronius Australia Pty. Ltd. *How to Set Up Export Limiting Using the Fronius Smart Meter*. White Paper. Version 1.0. Fronius Australia Pty. Ltd., 2015. URL: https://www.fronius.com/~/downloads/Solar%20Energy/Technical%20Articles/SE_TEA_Quick_Guide_How_to_set_up_Export_Limiting_using_the_Fronius_Smart_Meter_EN_AU.pdf.

[24] *Analysis of the Performance Indicators of the PV Power System*. Sydney Institute of Business and Technology, Western Sydney University, Sydney City Campus, Australia. 2018. DOI: https://doi.org/10.4236/jpee.2018.66005.

[25] ResearchHubs. *Power Output Variation with Wind Speed (Cut in/out speed)*. Accessed: 2025-05-04. 2015. URL: https://researchhubs.com/post/engineering/wind-energy/power-output-variation-with-wind-speed.html.

[26] Georges Hebrail and Alice Berard. *Individual Household Electric Power Consumption*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58K54. 2006.

[27] *15 Features of Python Every Developer Should Know — simplilearn.com*. https://www.simplilearn.com/python-features-article. [Accessed 22-05-2025].

[28] *pandas - Python Data Analysis Library — pandas.pydata.org*. https://pandas.pydata.org/. [Accessed 22-05-2025].

[29] *Pandas Pros and Cons Compared — altexsoft.com*. https://www.altexsoft.com/blog/pandas-library/. [Accessed 22-05-2025].

[30] *Home - PyCaret — pycaret.org*. https://pycaret.org/. [Accessed 22-05-2025].

[31] *AutoML | AutoML — automl.org*. https://www.automl.org/automl/. [Accessed 22-05-2025].

[32] *XGBoost — xgboost.ai*. https://xgboost.ai/. [Accessed 22-05-2025].

[33] *CatBoost - state-of-the-art open-source gradient boosting library with categorical features support — catboost.ai*. https://catboost.ai/. [Accessed 22-05-2025].

[34] *Welcome to LightGBM 2019;s documentation! 2014; LightGBM 4.6.0 documentation — lightgbm.readthedocs.io*. https://lightgbm.readthedocs.io/en/stable/. [Accessed 22-05-2025].

[35] *cvxpy.org*. https://www.cvxpy.org/index.html. [Accessed 22-05-2025].

[36] *Home — airflow.apache.org*. https://airflow.apache.org/. [Accessed 27-04-2025].

[37] EFET ENTSO-E and ebIX. *The Harmonised Electricity Market Role Model*. Tech. rep. Version 2023-01. Accessed May 22, 2025. ENTSO-E, EFET, and ebIX, 2023. URL: https://mwgstorage1.blob.core.windows.net/public/Ebix/Harmonised_Role_Model_2023-01.pdf.

[38] *PlantUML*. https://plantuml.com/. [Accessed 31-05-2025].

[39] Jin-Oh Lee and Yun-Su Kim. "Novel battery degradation cost formulation for optimal scheduling of battery energy storage systems". In: *International Journal of Electrical Power & Energy Systems* 137 (2022), p. 107795. ISSN: 0142-0615. DOI: https://doi.org/10.1016/j.ijepes.2021.107795. URL: https://www.sciencedirect.com/science/article/pii/S0142061521010140.

[40] *GitHub - AlDanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages. — github.com*. https://github.com/AlDanial/cloc. [Accessed 05-06-2025].

[41] John Forrest et al. *coin-or/Cbc: Release releases/2.10.12*. Version releases/2.10.12. Aug. 2024. DOI: 10.5281/zenodo.13347261. URL: https://doi.org/10.5281/zenodo.13347261.

[42] NVIDIA. *cuOpt - GPU accelerated Optimization Engine*. https://github.com/NVIDIA/cuopt. [Accessed 31-05-2025].

[43] Steven Diamond and Stephen Boyd. "CVXPY: A Python-embedded modeling language for convex optimization". In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[44] Akshay Agrawal et al. "A rewriting system for convex optimization problems". In: *Journal of Control and Decision* 5.1 (2018), pp. 42–60.

[45] GitHub User. *An in-process SQL OLAP database management system — duckdb.org*. https://duckdb.org/. [Accessed 22-05-2025].

[46] GitHub User. *SQL on Pandas — duckdb.org*. https://duckdb.org/docs/stable/guides/python/sql_on_pandas.html. [Accessed 22-05-2025].

[47] *Newsvendor problem - Cornell University Computational Optimization Open Textbook - Optimization Wiki — optimization.cbe.cornell.edu*. https://optimization.cbe.cornell.edu/index.php?title=Newsvendor_problem. [Accessed 03-05-2025].

Appendix A

Appendix

A.1 Code Repository

https://github.com/jokerofun/psa_tool

A.2 Microgrid Solutions

A.2.1 GPO-D Solution

Solution

```
1  def solve_microgrid_gpod(setup: MicrogridSetup):
2      problemClass = GraphProblemClass("microgrid_problem",
           time_length=setup.T)
3
4      metering_point = MeteringPoint("grid1")
5      homes = [Consumer(f"household_{i+1}") for i in range(setup.
           no_homes)]
6      school = Consumer("school")
7      solar_panels_homes = [SolarPanel(f"solar_panel_{i+1}") for i
           in range(setup.no_homes)]
8      solar_panel_school = SolarPanel("solar_panel_school")
9      wind_turbine = WindTurbine("wind_turbine")
10     batteries = [Battery(f"battery_{i+1}", setup.battery_power,
           setup.battery_power, setup.battery_capacity, setup.
           battery_efficiency) for i in range(setup.no_batteries)]
11
12     for home in homes:
```

```
13          home.dataflow.task("gen_consumption", DataProcessingTask,
                process_func=predict_consumer_data, parameters={"hours"
                : setup.T, "model_name":"consumer_model", "factor": 1},
                 final=True)
14      school.dataflow.task("gen_consumption", DataProcessingTask,
            process_func=predict_consumer_data, parameters={"hours":
            setup.T, "model_name":"consumer_model", "factor": 100},
            final=True)
15      for solar_panel in solar_panels_homes:
16          task1 = solar_panel.dataflow.task("get_solar_data",
                DataProcessingTask, process_func=get_irradiation_data,
                parameters={"latitude": 57.0488, "longitude": 9.9217})
17          task2 = solar_panel.dataflow.task("gen_solar_data",
                DataProcessingTask, process_func=
                generate_solar_panel_data, parameters={"rated_power":
                setup.solar_capacity_home}, final=True)
18          task1 >> task2
19      solar_school_task1 = solar_panel_school.dataflow.task("
            get_solar_data", DataProcessingTask, process_func=
            get_irradiation_data, parameters={"latitude": 57.0488, "
            longitude": 9.9217})
20      solar_school_task2 = solar_panel_school.dataflow.task("
            gen_solar_data", DataProcessingTask, process_func=
            generate_solar_panel_data, parameters={"rated_power": setup
            .solar_capacity_school}, final=True)
21      solar_school_task1 >> solar_school_task2
22      wind_task1 = wind_turbine.dataflow.task("get_wind_data",
            DataProcessingTask, process_func=get_wind_data, parameters
            ={"latitude": 57.0488, "longitude": 9.9217})
23      wind_task2 = wind_turbine.dataflow.task("gen_wind_data",
            DataProcessingTask, process_func=generate_wind_turbine_data
            , parameters={"rated_power": setup.wind_capacity, "
            cut_in_speed": 3.5, "rated_speed": 14, "cut_out_speed":
            25}, final=True)
24      wind_task1 >> wind_task2
25
26      problemClass.add_nodes([*homes, *solar_panels_homes, *
            batteries,
27                              school, solar_panel_school,
                                    wind_turbine,
28                              metering_point])
```

```
29
30      metering_point.connect_to([*homes, *solar_panels_homes, *
            batteries, school,
31                                 solar_panel_school, wind_turbine])
32
33      result = problemClass.solve(solver=cp.CBC, objective="minimize
            ", value="cost")
```

Listing A.1: GPO-D Microgrid Solution Implementation

Domain

```
1  class Consumer(Resource):
2      def __init__(self, name):
3          super().__init__(name)
4          self.consumption_kWh = []
5
6      def powerflow(self, t):
7          return -self.consumption_kWh[t]
8
9      @property
10     def variables(self):
11         return {self.name : {"powerFlow" : -self.consumption_kWh}}
12
13     def assign(self, t):
14         self.consumption_kWh = self.dataflow.results["
               gen_consumption"]["consumption_kWh"][:t]
15
16 class Producer(Resource):
17     def __init__(self, name):
18         super().__init__(name)
19         self.max_power_output_kW = []
20
21 class SolarPanel(Producer):
22     def __init__(self, name):
23         super().__init__(name)
24
25     def set_time_length(self, t):
26         self.c = cp.Variable(t, nonneg=True)
27
28     def powerflow(self, t):
29         return self.c[t] * self.max_power_output_kW[t]
```

```
30
31      def constraints(self, t):
32          return [self.c[t] <= 1]
33
34      @property
35      def variables(self):
36          return {self.name : {"production_schedule" : self.c.value
                * self.max_power_output_kW}}
37
38      def assign(self, t):
39          self.max_power_output_kW = self.dataflow.results["
                gen_solar_data"]["energy_generated"][:t]
40
41  class WindTurbine(Producer):
42      def __init__(self, name):
43          super().__init__(name)
44
45      def powerflow(self, t):
46          return self.max_power_output_kW[t]
47
48      @property
49      def variables(self):
50          return {self.name : {"production_schedule" : self.
                max_power_output_kW}}
51
52      def assign(self, t):
53          self.max_power_output_kW = self.dataflow.results["
                gen_wind_data"]["energy_generated"][:t]
54
55  class Battery(Resource):
56      def __init__(self, name, charging_power_kW,
            discharging_power_kW, capacity_kWh, efficiency):
57          super().__init__(name)
58          self.charging_power_kW = charging_power_kW
59          self.discharging_power_kW = discharging_power_kW
60          self.capacity_kWh = capacity_kWh
61          self.efficiency = efficiency
62
63      def set_time_length(self, t):
64          self.charge = cp.Variable(t, nonneg=True)
65          self.discharge = cp.Variable(t, nonneg=True)
```

```
66          self.SoC = cp.Variable(shape = (t+1), nonneg=True)
67
68      def const_constraints(self):
69          return [self.SoC[0] == 0]
70
71      def constraints(self, t):
72          constraints = [
73              self.SoC[t+1] <= self.capacity_kWh,
74              self.charge[t] <= self.charging_power_kW,
75              self.discharge[t] <= self.discharging_power_kW,
76              self.SoC[t+1] == self.SoC[t] + self.efficiency * self.
                    charge[t] - (1 / self.efficiency) * self.discharge[
                    t]
77          ]
78          return constraints
79
80      def powerflow(self, t):
81          return self.discharge[t] - self.charge[t]
82
83      @property
84      def variables(self):
85          return {self.name : {"SOC" : self.SoC.value, "powerFlow":
                self.discharge.value - self.charge.value}}
86
87  class MeteringPoint(Resource):
88      def __init__(self, name):
89          super().__init__(name)
90          # self.connect_nodes([self])
91
92      def set_time_length(self, t):
93          self.energy_import = cp.Variable(t, nonneg=True)
94
95      def powerflow(self, t):
96          return self.energy_import[t]
97
98      @property
99      def cost(self):
100         return cp.sum(self.energy_import)
101
102     @property
103     def variables(self):
```

```
104         return {self.name : {"powerFlow" : self.energy_import.
                value}}
```

Listing A.2: GPO-D Microgrid Domain Implementation

A.2.2 CVXPY Solution

```
1  def solve_microgrid_cvxpy(setup:MicrogridSetup):
2      home_demands = []
3      for _ in range(setup.no_homes):
4          home_demand_dict = predict_consumer_data(dataframe={},
               parameters={"hours": setup.T, "model_name":"
               consumer_model", "factor": 1})
5          home_demand = home_demand_dict["gen_consumption"]["
               consumption_kWh"]
6          home_demands.append(home_demand)
7      school_demand_dict = predict_consumer_data(dataframe={},
           parameters={"hours": setup.T, "model_name":"consumer_model"
           , "factor": 100})
8      school_demand = school_demand_dict["gen_consumption"]["
           consumption_kWh"]
9      total_demand = [sum(group) for group in zip(*home_demands)] +
           school_demand
10
11     wind_data_dict = get_wind_data({}, parameters={"latitude":
           57.0488, "longitude": 9.9217})
12     wind_prod_dict = generate_wind_turbine_data(wind_data_dict,
           parameters={"rated_power": setup.wind_capacity, "
           cut_in_speed": 3.5, "rated_speed" : 14, "cut_out_speed":
           25})
13     wind_prod = wind_prod_dict["gen_wind_data"]["energy_generated"
           ]
14
15     solar_data_dict = get_irradiation_data({}, parameters={"
           latitude": 57.0488, "longitude": 9.9217})
16     solar_prod_school_dict = generate_solar_panel_data(
           solar_data_dict, parameters={"rated_power": setup.
           solar_capacity_school})
17     solar_prod_school = solar_prod_school_dict["gen_solar_data"]["
           energy_generated"]
18
```

```
19      solar_prods = []
20      for _ in range(setup.no_solar_panels):
21          solar_data_dict =  get_irradiation_data({}, parameters={"
               latitude": 57.0488, "longitude": 9.9217})
22          solar_prod_dict = generate_solar_panel_data(
               solar_data_dict, parameters={"rated_power": setup.
               solar_capacity_home,})
23          solar_prod = solar_prod_dict["gen_solar_data"]["
               energy_generated"]
24          solar_prods.append(solar_prod)
25
26      grid_import = cp.Variable(setup.T, nonneg=True)
27      battery_charge = [cp.Variable(setup.T, nonneg=True) for _ in
           range(setup.no_batteries)]
28      battery_discharge = [cp.Variable(setup.T, nonneg=True) for _
           in range(setup.no_batteries)]
29      battery_soc = [cp.Variable(setup.T+1, nonneg=True) for _ in
           range(setup.no_batteries)]
30      c_homes = [cp.Variable(setup.T, nonneg=True) for _ in range(
           setup.no_homes)]
31      c_school = cp.Variable(setup.T, nonneg=True)
32
33      constraints = []
34      for i in range(setup.no_batteries):
35          constraints.append(battery_soc[i][0] == 0)
36      for t in range(setup.T):
37          for i in range(setup.no_homes):
38              constraints.append(c_homes[i][t] <= 1)
39          constraints.append(c_school[t] <= 1)
40
41          total_hourly_battery_discharge = sum(battery_discharge[i][
               t] for i in range(setup.no_batteries))
42          total_hourly_battery_charge = sum(battery_charge[i][t] for
               i in range(setup.no_batteries))
43          total_hourly_solar_prod_homes = sum(c_homes[i][t] *
               solar_prods[i][t] for i in range(setup.no_solar_panels)
               )
44          total_hourly_solar_prod_school = c_school[t] *
               solar_prod_school[t]
45          constraints.append(
46              grid_import[t] + total_hourly_battery_discharge +
```

```
                total_hourly_solar_prod_homes +
                total_hourly_solar_prod_school + wind_prod[t]
47              == total_hourly_battery_charge + total_demand[t])
48
49          for i in range(setup.no_batteries):
50              constraints.append(battery_charge[i][t] <= setup.
                    battery_power)
51              constraints.append(battery_discharge[i][t] <= setup.
                    battery_power)
52
53              constraints.append(
54                  battery_soc[i][t+1] == battery_soc[i][t] +
                        battery_charge[i][t] *
55                  setup.battery_efficiency - battery_discharge[i][t]
                        / setup.battery_efficiency)
56
57              constraints.append(battery_soc[i][t+1] <= setup.
                    battery_capacity)
58
59      objective = cp.Minimize(cp.sum(grid_import))
60      prob = cp.Problem(objective, constraints)
61      prob.solve(solver=cp.CBC, verbose=False)
```

Listing A.3: CVXPY Microgrid Solution Implementation

A.2.3 Pyomo Solution

```
1   def solve_microgrid_pyomo(setup:MicrogridSetup):
2       home_demands = []
3       for _ in range(setup.no_homes):
4           home_demand_dict = predict_consumer_data(dataframe={},
                parameters={"hours": setup.T, "model_name":"
                consumer_model", "factor": 1})
5           home_demand = home_demand_dict["gen_consumption"]["
                consumption_kWh"]
6           home_demands.append(home_demand)
7       school_demand_dict = predict_consumer_data(dataframe={},
            parameters={"hours": setup.T, "model_name":"consumer_model"
            , "factor": 100})
8       school_demand = school_demand_dict["gen_consumption"]["
            consumption_kWh"]
```

```
9       total_demand = [sum(group) for group in zip(*home_demands)] +
            school_demand
10
11      wind_data_dict = get_wind_data({}, parameters={"latitude":
            57.0488, "longitude": 9.9217})
12      wind_prod_dict = generate_wind_turbine_data(wind_data_dict,
            parameters={"rated_power": setup.wind_capacity, "
            cut_in_speed": 3.5, "rated_speed" : 14, "cut_out_speed":
            25})
13      wind_prod = wind_prod_dict["gen_wind_data"]["energy_generated"
            ]
14
15      solar_data_dict = get_irradiation_data({}, parameters={"
            latitude": 57.0488, "longitude": 9.9217})
16      solar_prod_school_dict = generate_solar_panel_data(
            solar_data_dict, parameters={"rated_power": setup.
            solar_capacity_school})
17      solar_prod_school = solar_prod_school_dict["gen_solar_data"]["
            energy_generated"]
18
19      solar_prods = []
20      for _ in range(setup.no_solar_panels):
21          solar_data_dict =  get_irradiation_data({}, parameters={"
                latitude": 57.0488, "longitude": 9.9217})
22          solar_prod_dict = generate_solar_panel_data(
                solar_data_dict, parameters={"rated_power": setup.
                solar_capacity_home,})
23          solar_prod = solar_prod_dict["gen_solar_data"]["
                energy_generated"]
24          solar_prods.append(solar_prod)
25
26      model = pyo.ConcreteModel()
27      model.T = pyo.RangeSet(0, setup.T-1)
28      model.B = pyo.RangeSet(0, setup.no_batteries-1)
29      model.H = pyo.RangeSet(0, setup.no_homes-1)
30      model.S = pyo.RangeSet(0, setup.no_big_solar_panels-1)
31
32      model.grid_import = pyo.Var(model.T, domain=pyo.
            NonNegativeReals)
33      model.battery_charge = pyo.Var(model.B, model.T, domain=pyo.
            NonNegativeReals)
```

```
34      model.battery_discharge = pyo.Var(model.B, model.T, domain=pyo
            .NonNegativeReals)
35      model.battery_soc = pyo.Var(model.B, range(setup.T+1), domain=
            pyo.NonNegativeReals)
36      model.c_home = pyo.Var(model.H, model.T, domain=pyo.
            NonNegativeReals)
37      model.c_school = pyo.Var(model.T, domain=pyo.NonNegativeReals)
38
39      def soc_init_rule(m, b):
40          return m.battery_soc[b,0] == 0
41      model.soc_init = pyo.Constraint(model.B, rule=soc_init_rule)
42
43      def power_balance_rule(m, t):
44          total_hourly_battery_discharge = sum(m.battery_discharge[b
                , t] for b in model.B)
45          total_hourly_battery_charge = sum(m.battery_charge[b, t]
                for b in model.B)
46          total_hourly_solar_prod_homes = sum(m.c_home[h, t] *
                solar_prods[h][t] for h in model.H)
47          total_hourly_solar_prod_school = m.c_school[t] *
                solar_prod_school[t]
48          return (m.grid_import[t] + total_hourly_solar_prod_homes +
                 total_hourly_solar_prod_school + wind_prod[t] +
                total_hourly_battery_discharge ==
49                   total_hourly_battery_charge + total_demand[t])
50      model.power_balance = pyo.Constraint(model.T, rule=
            power_balance_rule)
51
52      def battery_charge_limit_rule(m, b, t):
53          return m.battery_charge[b, t] <= setup.battery_power
54      model.battery_charge_limit = pyo.Constraint(model.B, model.T,
            rule=battery_charge_limit_rule)
55
56      def battery_discharge_limit_rule(m, b, t):
57          return m.battery_discharge[b, t] <= setup.battery_power
58      model.battery_discharge_limit = pyo.Constraint(model.B, model.
            T, rule=battery_discharge_limit_rule)
59
60      def soc_update_rule(m, b, t):
61          return m.battery_soc[b, t+1] == m.battery_soc[b, t] + m.
                battery_charge[b, t] * setup.battery_efficiency - m.
```

```
            battery_discharge[b, t] / setup.battery_efficiency
62      model.soc_update = pyo.Constraint(model.B, range(setup.T),
            rule=soc_update_rule)
63
64      def soc_max_rule(m, b, t):
65          return m.battery_soc[b, t+1] <= setup.battery_capacity
66      model.soc_max = pyo.Constraint(model.B, range(setup.T), rule=
            soc_max_rule)
67
68      def c_home_max_rule(m, h, t):
69          return m.c_home[h, t] <= 1
70      model.c_home_max = pyo.Constraint(model.H, range(setup.T),
            rule=c_home_max_rule)
71
72      def c_school_max_rule(m, t):
73          return m.c_school[t] <= 1
74      model.c_school_max = pyo.Constraint(range(setup.T), rule=
            c_school_max_rule)
75
76      model.obj = pyo.Objective(expr=sum(model.grid_import[t] for t
            in model.T), sense=pyo.minimize)
77
78      solver = pyo.SolverFactory('cbc', executable="examples/Pyomo/
            cbc/bin/cbc.exe")
79      result = solver.solve(model, tee=False)
```

Listing A.4: Pyomo Microgrid Solution Implementation

A.2.4 GBOML Solution

Solution

```
1  def solve_microgrid_gboml(setup:MicrogridSetup,
       microgrid_file_path="examples/GBOML/microgrid.txt"):
2      setup.T += 1
3      gboml_domain.build_microgrid(setup=setup, file_path=
           microgrid_file_path)
4
5      home_demands = []
6      for _ in range(setup.no_homes):
7          home_demand_dict = predict_consumer_data(dataframe={},
               parameters={"hours": setup.T, "model_name":"
```

```
            consumer_model", "factor": 1})
8           home_demand = home_demand_dict["gen_consumption"]["
                consumption_kWh"]
9           home_demands.append(home_demand)
10      school_demand_dict = predict_consumer_data(dataframe={},
            parameters={"hours": setup.T, "model_name":"consumer_model"
            , "factor": 100})
11      school_demand = school_demand_dict["gen_consumption"]["
            consumption_kWh"]
12      demand =  [sum(group) for group in zip(*home_demands)] +
            school_demand
13      np.savetxt("data/gboml_data/demand.csv", demand)
14
15      wind_data_dict = get_wind_data({}, parameters={"latitude":
            57.0488, "longitude": 9.9217})
16      wind_prod_dict = generate_wind_turbine_data(wind_data_dict,
            parameters={"rated_power": setup.wind_capacity, "
            cut_in_speed": 3.5, "rated_speed" : 14, "cut_out_speed":
            25})
17      wind_prod = wind_prod_dict["gen_wind_data"]["energy_generated"
            ]
18      np.savetxt("data/gboml_data/gen_wind.csv", wind_prod[:setup.T
            ])
19
20      solar_data_dict = get_irradiation_data({}, parameters={"
            latitude": 57.0488, "longitude": 9.9217})
21      solar_prod_school_dict = generate_solar_panel_data(
            solar_data_dict, parameters={"rated_power": setup.
            solar_capacity_school})
22      solar_prod_school = solar_prod_school_dict["gen_solar_data"]["
            energy_generated"]
23      np.savetxt("data/gboml_data/gen_big_solar.csv",
            solar_prod_school[:setup.T])
24
25      solar_prod = {}
26      for i in range(1, setup.no_solar_panels + 1):
27          solar_data_dict = get_irradiation_data(dataframe={},
                parameters={"latitude": 57.0488, "longitude": 9.9217})
28          solar_prod_dict = generate_solar_panel_data(
                solar_data_dict, parameters={"rated_power": setup.
                solar_capacity_home,})
```

```
29            solar_prod = solar_prod_dict["gen_solar_data"]["
                  energy_generated"]
30            np.savetxt(f"data/gboml_data/gen_solar_{i}.csv",
                  solar_prod[:setup.T])
31
32        gboml_model = GbomlGraph(setup.T)
33        nodes, edges, global_params = gboml_model.
              import_all_nodes_and_edges(microgrid_file_path)
34        gboml_model.add_nodes_in_model(*nodes)
35        gboml_model.add_hyperedges_in_model(*edges)
36        gboml_model.build_model()
37
38        solution = gboml_model.solve_clp()
39        setup.T -= 1
```

Listing A.5: GBOML Microgrid Solution Implementation

Domain

```
1  #TIMEHORIZON
2  T = 24+1;
3
4  #NODE DEMAND
5  #PARAMETERS
6  total_demand = import "../../data/gboml_data/demand.csv";
7  #VARIABLES
8  external: consumption[T];
9  #CONSTRAINTS
10 consumption[t] == total_demand[t];
11
12 #NODE SOLAR_PV_1
13 #PARAMETERS
14 max_power_output_kW = import "../../data/gboml_data/gen_solar_1.
       csv";
15 #VARIABLES
16 internal: c[T];
17 external: electricity[T];
18 #CONSTRAINTS
19 c[t] >= 0;
20 c[t] <= 1;
21 electricity[t] >= 0;
22 electricity[t] == c[t] * max_power_output_kW[t];
```

```
23
24 #NODE SOLAR_PV_2
25 #PARAMETERS
26 max_power_output_kW = import "../../data/gboml_data/gen_solar_2.
       csv";
27 #VARIABLES
28 internal: c[T];
29 external: electricity[T];
30 #CONSTRAINTS
31 c[t] >= 0;
32 c[t] <= 1;
33 electricity[t] >= 0;
34 electricity[t] == c[t] * max_power_output_kW[t];
35
36 .
37 .
38 .
39
40 #NODE SOLAR_PV_50
41 #PARAMETERS
42 max_power_output_kW = import "../../data/gboml_data/gen_solar_50.
       csv";
43 #VARIABLES
44 internal: c[T];
45 external: electricity[T];
46 #CONSTRAINTS
47 c[t] >= 0;
48 c[t] <= 1;
49 electricity[t] >= 0;
50 electricity[t] == c[t] * max_power_output_kW[t];
51
52 #NODE BIG_SOLAR_PV_1
53 #PARAMETERS
54 max_power_output_kW = import "../../data/gboml_data/gen_big_solar.
       csv";
55 #VARIABLES
56 internal: c[T];
57 external: electricity[T];
58 #CONSTRAINTS
59 c[t] >= 0;
60 c[t] <= 1;
```

```
electricity[t] >= 0;
electricity[t] == c[t] * max_power_output_kW[t];

#NODE WIND_TURBINE_1
#PARAMETERS
max_power_output_kW = import "../../data/gboml_data/gen_wind.csv";
#VARIABLES
external: electricity[T];
#CONSTRAINTS
electricity[t] >= 0;
electricity[t] == max_power_output_kW[t];

#NODE BATTERY_1
#PARAMETERS
charging_power_kW = 500;
discharging_power_kW = 500;
capacity_kWh = 500;
efficiency = 0.9;
SoC = 0;
#VARIABLES
internal: energy[T];
external: charge[T];
external: discharge[T];
#CONSTRAINTS
energy[t] >= 0;
charge[t] >= 0;
discharge[t] >= 0;
energy[t] <= capacity_kWh;
charge[t] <= charging_power_kW;
discharge[t] <= discharging_power_kW;
energy[t+1] == energy[t] + efficiency * charge[t] - discharge[t] /
    efficiency;
energy[0] == SoC;

#NODE BATTERY_2
#PARAMETERS
charging_power_kW = 500;
discharging_power_kW = 500;
capacity_kWh = 500;
efficiency = 0.9;
SoC = 0;
```

```
101 #VARIABLES
102 internal: energy[T];
103 external: charge[T];
104 external: discharge[T];
105 #CONSTRAINTS
106 energy[t] >= 0;
107 charge[t] >= 0;
108 discharge[t] >= 0;
109 energy[t] <= capacity_kWh;
110 charge[t] <= charging_power_kW;
111 discharge[t] <= discharging_power_kW;
112 energy[t+1] == energy[t] + efficiency * charge[t] - discharge[t] /
        efficiency;
113 energy[0] == SoC;
114
115 #NODE BATTERY_3
116 #PARAMETERS
117 charging_power_kW = 500;
118 discharging_power_kW = 500;
119 capacity_kWh = 500;
120 efficiency = 0.9;
121 SoC = 0;
122 #VARIABLES
123 internal: energy[T];
124 external: charge[T];
125 external: discharge[T];
126 #CONSTRAINTS
127 energy[t] >= 0;
128 charge[t] >= 0;
129 discharge[t] >= 0;
130 energy[t] <= capacity_kWh;
131 charge[t] <= charging_power_kW;
132 discharge[t] <= discharging_power_kW;
133 energy[t+1] == energy[t] + efficiency * charge[t] - discharge[t] /
        efficiency;
134 energy[0] == SoC;
135
136 #NODE METERING_POINT
137 #VARIABLES
138 external: power_import[T];
139 #CONSTRAINTS
```

```
power_import[t] >= 0;
#OBJECTIVES
min: power_import[t];

#HYPEREDGE POWER_BALANCE
#CONSTRAINTS
METERING_POINT.power_import[t] + SOLAR_PV_1.electricity[t] +
    SOLAR_PV_2.electricity[t] + ... + SOLAR_PV_50.electricity[t] +
    BIG_SOLAR_PV_1.electricity[t] + WIND_TURBINE_1.electricity[t] +
     BATTERY_1.discharge[t] + BATTERY_2.discharge[t] + BATTERY_3.
    discharge[t] == DEMAND.consumption[t] + BATTERY_1.charge[t] +
    BATTERY_2.charge[t] + BATTERY_3.charge[t];
```

Listing A.6: GBOML Microgrid Domain Implementation

A.3 Comparison Experiments Results

A.3.1 Optimization Problem Insights

Solution	Variables	Extra	Time Horizon	Homes	Solar Panels	Total Variables Size
GPO-D	61	3	24	50	50	1467
CVXPY	61	3	24	50	50	1467
Pyomo	61	3	24	50	50	1467
GBOML	114	3	24	50	50	2739
GPO-D	61	3	48	50	50	2931
CVXPY	61	3	48	50	50	2931
Pyomo	61	3	48	50	50	2931
GBOML	114	3	48	50	50	5475
GPO-D	61	3	72	50	50	4395
CVXPY	61	3	72	50	50	4395
Pyomo	61	3	72	50	50	4395
GBOML	114	3	72	50	50	8211
GPO-D	61	3	120	50	50	7323
CVXPY	61	3	120	50	50	7323
Pyomo	61	3	120	50	50	7323
GBOML	114	3	120	50	50	13683
GPO-D	111	3	24	100	100	2667
CVXPY	111	3	24	100	100	2667
Pyomo	111	3	24	100	100	2667
GBOML	214	3	24	100	100	5139
GPO-D	211	3	24	200	200	5067
CVXPY	211	3	24	200	200	5067
Pyomo	211	3	24	200	200	5067
GBOML	414	3	24	200	200	9939
GPO-D	511	3	24	500	500	12267
CVXPY	511	3	24	500	500	12267
Pyomo	511	3	24	500	500	12267
GBOML	-	-	24	500	500	-
GPO-D	1011	3	24	1000	1000	24267
CVXPY	1011	3	24	1000	1000	24267
Pyomo	1011	3	24	1000	1000	24267
GBOML	-	-	24	1000	1000	-

Table A.1: Number of variables for the microgrid optimization problem

Solution	Constraints	Extra	Time Horizon	Homes	Solar Panels	Total Constraints Size
GPO-D	125	3	24	50	50	3003
CVXPY	125	3	24	50	50	3003
Pyomo	125	3	24	50	50	3003
GBOML	230	3	24	50	50	5523
GPO-D	125	3	48	50	50	6003
CVXPY	125	3	48	50	50	6003
Pyomo	125	3	48	50	50	6003
GBOML	230	3	48	50	50	11043
GPO-D	125	3	72	50	50	9003
CVXYP	125	3	72	50	50	9003
Pyomo	125	3	72	50	50	9003
GBOML	230	3	72	50	50	16563
GPO-D	125	3	120	50	50	15003
CVXPY	125	3	120	50	50	15003
Pyomo	125	3	120	50	50	15003
GBOML	230	3	120	50	50	27603
GPO-D	225	3	24	100	100	5403
CVXPY	225	3	24	100	100	5403
Pyomo	225	3	24	100	100	5403
GBOML	430	3	24	100	100	10323
GPO-D	425	3	24	200	200	10203
CVXPY	425	3	24	200	200	10203
Pyomo	425	3	24	200	200	10203
GBOML	830	3	24	200	200	19923
GPO-D	1025	3	24	500	500	24603
CVXPY	1025	3	24	500	500	24603
Pyomo	1025	3	24	500	500	24603
GBOML	-	-	24	500	500	-
GPO-D	2025	3	24	1000	1000	48603
CVXPY	2025	3	24	1000	1000	48603
Pyomo	2025	3	24	1000	1000	48603
GBOML	-	-	24	1000	1000	-

Table A.2: Number of constraints for the microgrid optimization problem

A.3.2 Productivity Eexperiments

Implementation	Characters (with whitespaces)	LOC	ELOC
GPOD	2395 (2769)	32	28
GPOD (domain)	2448 (3231)	104	78
GPOD (all)	4843 (6000)	136	106
CVXPY	3125 (3721)	61	51
PYOMO	3957 (4630)	78	64
GBOML	2128 (2430)	38	33
GBOML (txt file)	15137 (17739)	765	—
GBOML (including txt file)	17265 (20169)	803	—

Table A.3: Character count, Lines of Code (LOC), and Effective Lines of Code (ELOC) for different microgrid problem implementations.