

Bachelor Thesis

**Developing a method for comparing free
open source energy supply system
modelling software in the context of
computational complexity**

Frederik Emmel

June 2022

Bachelor Thesis**Developing a method for comparing free
open source energy supply system
modelling software in the context of
computational complexity****Frederik Emmel**

Matr.-Nr.: 21652458

Erstprüfer: Dr.-Ing. Kristin Abel-Günther

Zweitprüfer: Mathias Ammon M.Sc.

Betreuer: Mathias Ammon M.Sc.

Hamburg, 30. June 2022

TUHH

Technische Universität Hamburg

**Institut für
Energietechnik**

PROF. DR.-ING. A. KATHER

Bachelor Thesis for Mr. Frederik Emmel

Matr.-Nr. 21652458

**Developing a method for comparing free open source energy
supply system modelling software in the context of
computational complexity**

TUHH

Technische Universität Hamburg

Figure 0.1: Logo of the TUHH

(Dr.-Ing. Kristin Abel-Günther)

Ich erkläre hiermit, dass die vorliegende Bachelor Thesis ohne fremde Hilfe selbstständig verfasst wurde und nur die angegebenen Quellen und Hilfsmittel benutzt worden sind. Wörtlich oder sinngemäß aus anderen Werken entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Alle Quellen, die dem World Wide Web entnommen oder in einer sonstigen digitalen Form verwendet wurden, sind der Arbeit beigefügt.

Diese Arbeit ist nach bestem Wissen erstellt worden. Für den Inhalt kann jedoch keine Gewähr übernommen werden.

Hamburg, 30. June 2022


(Frederik Emmel)

Contents

List of Figures	III
List of Tables	V
1 Introduction	1
2 Theoretical Framework	3
2.1 Energy Systems	3
2.2 Energy system modelling	4
2.3 Tessif	5
2.4 Oemof	6
2.5 PyPSA	7
2.6 Computational complexity theory	7
2.7 Computational complexity	7
2.8 Time complexity	8
2.9 Space complexity	8
2.10 Scalability	8
2.11 Python	8
2.12 Time and space measurement	9
2.13 Time libraries	9
2.13.1 Time library	9
2.13.2 Timeit library	9
2.13.3 Decision on the time measurement library	10
2.14 Memory measurment libraries	10
2.14.1 tracemalloc	10
2.14.2 psutil	10
2.14.3 Decision on memory library	10
2.15 Implementing functions into Tessif	10
2.16 Time measurement	11
2.17 Memory space measurement function	12
2.18 Functions to give a quick overview of Energy Systems simulation	12
2.18.1 stop time function	12
2.18.2 trace memory function	13
2.19 Visualization of stop time and trace memory	13
2.20 Scalability	14
2.20.1 Creation of a minimum self similar energy system	14
2.20.2 Creation of a self similar energy system	15
2.20.3 asses scalability	16
2.21 Visualization of scalability functions	18
2.21.1 2D visualization	18
2.21.2 3D visualization	19
2.22 Bachmann-Landau symbols	20

3	Evaluation	21
3.1	Verification of the collected data:	21
3.1.1	Measure of dispersion	21
3.1.2	Calculation of the measure of dispersion	22
3.2	Analysis	23
3.3	Scalability regarding the timescale	24
3.3.1	Expectation	24
3.3.2	Oemof time complexity	24
3.3.3	PyPSA time complexity	25
3.3.4	Oemof memory complexity	26
3.3.5	PyPSA memory complexity	27
3.3.6	Comparison of Oemof and PyPSA time and space	28
3.4	Scalability regarding the size of the energy system	30
3.4.1	Expectation	30
3.4.2	Oemof time complexity	31
3.4.3	PyPSA time complexity	32
3.4.4	Oemof space complexity	33
3.4.5	PyPSA space complexity	34
3.4.6	Comparison of Oemof and PyPSA time and space	36
3.5	comparison timescale and scale of the energy system	39
3.5.1	Results	39
4	Conclusion and Outlook	43
4.1	Conclusion	43
4.2	Outlook	43
	Bibliography	44

List of Figures

0.1	Logo of the TUHH	
2.1	Graphical display of an energy system (own figure)	4
2.2	Representation of an Energy system as a graph(own figure generated with Tessif)	6
2.3	bar plot of the time visualization function (own figure created with Tessif)	14
2.4	minimum self similar energy system (own figure created with Tessif) . . .	15
2.5	Two minimum self similar energy system connected into a self similar energy system (own figure created with Tessif)	16
2.6	Flow chart of the asses_scalability function (own figure)	17
2.7	2D visualization of the energy systems optimized with oemof	18
2.8	Stacked 3D bar plot of a simulation with Oemof	19
3.1	calculation of the standard variation (own figure)	23
3.2	results of the Oemof time measurement (own figure created with Tessif) .	25
3.3	results of the PyPSA time measurement (own figure created with Tessif) .	26
3.4	results of the Oemof memory usage measurement (own figure created with Tessif)	27
3.5	results of the PyPSA memory usage measurement (own figure created with Tessif)	28
3.6	Comparison of the optimization process of the modelling tools Oemof and PyPSA (own figure)	29
3.7	Comparison of the post-processing process of the modelling tools Oemof and PyPSA (own figure)	30
3.8	Development of the time consumption by enhancing the energy system size (own figure)	31
3.9	Development of the time consumption for every step when enhancing the energy system size (own figure)	32
3.10	Development of the time consumption for every step when enhancing the energy system size (own figure)	33
3.11	Development of the time consumption by enhancing the energy system size (own figure)	33
3.12	Development of the time consumption by enhancing the energy system size (own figure)	34
3.13	Development of the time consumption by enhancing the energy system size (own figure)	34
3.14	Development of the memory usage when enhancing the energy system size (own figure)	35
3.15	Development of the memory usage when enhancing the energy system size (own figure)	36
3.16	Comparison of the memory usage of the third step (own figure)	37
3.17	Comparison of the memory usage of the third step (own figure)	38

3.18	Comparison of the memory usage of the third step (own figure)	38
3.19	stacked 3D plot - Oemof (own figure created with Tessif)	40
3.20	stacked 3D plot - PyPSA (own figure created with Tessif)	40
3.21	time measurement bar plot - Oemof (own figure created with Tessif) . . .	41
3.22	time measurement bar plot - PyPSA (own figure created with Tessif) . . .	42

List of Tables

1 Introduction

For several decades, scientists have warned about the dangers of climate change [1]. Today, extreme weather events such as very high temperatures or flooding as a result of the anthropogenic climate change are already visible [2]. Without more restrictive measures to mitigate greenhouse gases, the global mean temperature will increase by 3.7-4.8 degrees Celsius compared to pre industrial times according to the IPCC [3].

Germany wants to reduce its greenhouse gas emissions by 65% until 2030 and aims to be net zero by 2045 [4]. The largest greenhouse gas emitter was detected to be the energy supply sector. On a global scale, the emissions of this sector are predicted to double or even triple until 2050 compared to those 2010, unless new ways of mitigation are introduced [3]. In order to keep the rise in temperature under the 1.5 degrees Celsius agreed on the Paris agreement, the total CO₂ emissions of this sector have to decline by 45% from the level of 2010 and have to be net zero by 2050 [3].

In order to achieve this, there needs to be a shift from fossil fuels to renewable energies. This change from a few centralized energy sources to many decentralized renewable energy sources is a huge economic and organisational challenge. Obviously, huge investments and many difficult political decisions have to be made. Energy system modelling can play a crucial role in guiding these investments and decisions.

There already are various energy system modelling tools, some of them commercial licensed and some of them open source. However, (free) open source programs are of particular significance, because it is important that these difficult decisions are made based on information accessible to everyone. In addition, since the trend goes to systems of decentralized renewable energy sources, economic considerations play a growing role. This is in particular true for poorer regions of the world.

To lower the threshold of using these software, the Hamburg University of Technology started a program called Tessif (short for Transforming Energy Supply System Framework). Through this framework it is possible to use different free open source energy system modelling software.

One of the crucial characteristics of an algorithm, or in this particular case a modelling tool, is the computational complexity, since time (or memory space) is an extremely valuable resource. The main objective of this thesis is to develop a method to measure and compare the computational complexity of open source energy system modelling tools used through Tessif. We will use this tool to then analyse and compare two commonly used modelling tools, namely Oemof and PyPSA.

To do so, we start by giving an overview of the theoretical background and describe the development of the measurement functions. Subsequently follows the above mentioned analysis of the two open source modelling tools. Finally, we summarize the results. Most importantly, it is verified that the complexity class for the total resource usage as well as for the running time and the memory usage is $\mathcal{O}(n)$ for both modelling tools.

Moreover, the biggest resource consumer is the optimization process which is done by the integrated modelling tool.

2 Theoretical Framework

The purpose of the following chapter is to build the theoretical framework for this thesis. We start by discussing the general principles of energy systems and energy system modelling. Following that, various tools for energy system modelling are introduced. We focus on open source software because of their special significance. Most importantly, Tessif and in particular its ability to execute several different simulation tools are reviewed.

Finally, in view of the main objective of this thesis, we introduce the concept of computational complexity of an algorithm. This chapter is then concluded by an throughout review of our functions, which we built to analyse the computational complexity of modelling tools in Tessif.

2.1 Energy Systems

An energy system is defined as a system that comprises all processes from the energy generation to the final consumption of the end user [3].

The first component of an energy system is the primary energy, for example oil, radioactive plutonium or wind. In order to use this primary energy it has to be processed, e.g. oil is refined, plutonium is used in nuclear power plants and wind is transformed with help of turbines into electricity. This is called the conversion to secondary energy. From there, the energy is either transported to the place of final consumption or it is stored. In our example, the generated electricity of either the nuclear power plant or turbines is fed into an electrical grid from where it can be stored in a storage system or delivered to the end consumer. The refined oil on the other hand is directly transported to the place of final consumption. In Figure 2.1 the process from the primary energy to the final consumption or storage for the usual forms of primary energy is summarized.

There are several ways to store energy, ranging from batteries to hydropower storages. From there the energy can be resupplied to the electricity grid if the demand is higher than the current energy production. However, with every conversion there are losses. Indeed, this is a big issue in the energy supply sector, which loses 29,3% of the primary energy given into the systems. Therefore, one could say that the energy supply sector itself is its biggest consumer [3] .

The energy supply sector is the largest sectorwise contributor of greenhouse gases. It is responsible for about 35% of the total anthropogenic emissions in 2010. In addition, the energy demand is growing fast. The gradient grew from 1.7% a year from 1990-2000 to 3.1% from 2000-2010. Consequently, it is not surprising that the baseline scenarios from the IPCC predict that the annual emissions from the energy supply sector will grow from 14.4 GtCO₂/perYear in 2010 to 24-33 GtCO₂/perYear in 2050 [3].

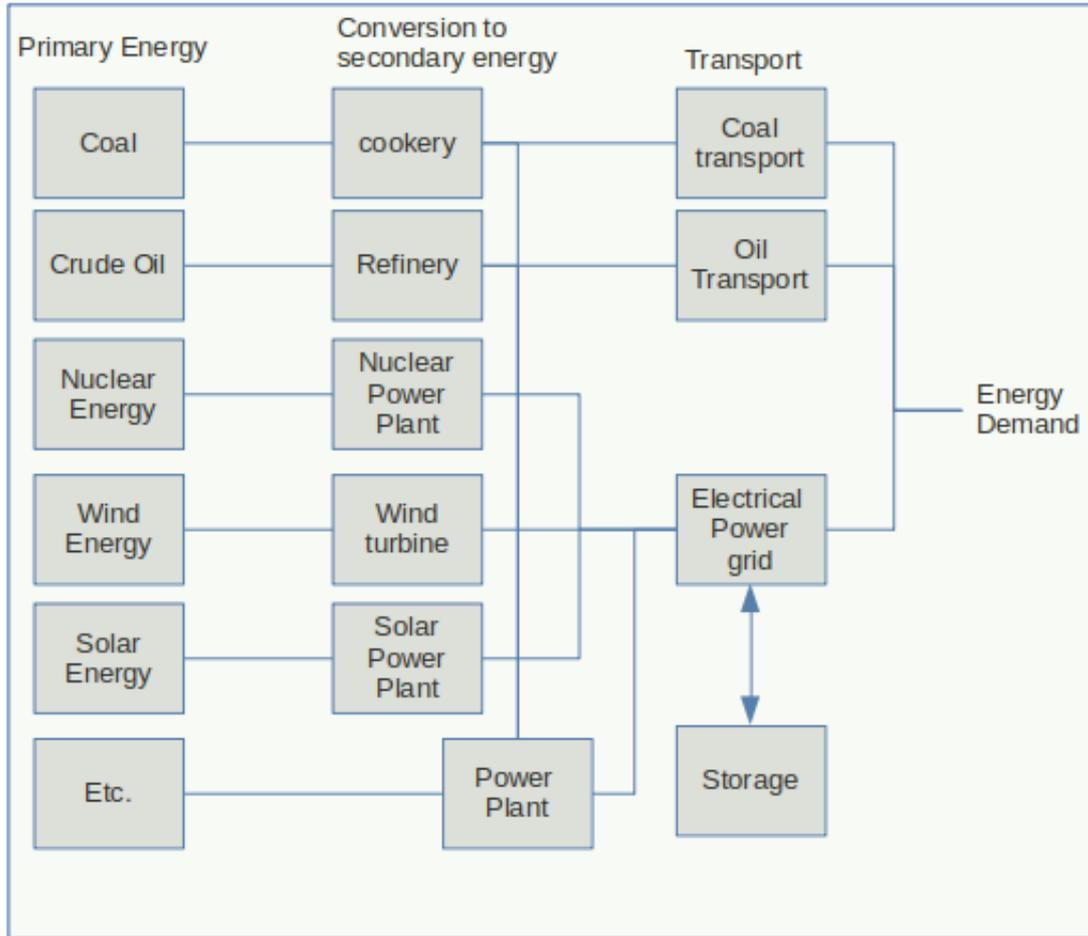


Figure 2.1: Graphical display of an energy system (own figure)

Consequently, good ideas on how to mitigate emissions by the energy supply sector are essential for battling climate change. Improving on the energy efficiency of fossil power plants or change from coal to gas is not going to be sufficient to realise the goals agreed on in the Paris agreement. There needs to be a fundamental transformation of the energy supply sector. Energy system modelling is expected to play an important role in this fundamental transformation, in a way outlined in the next section.

2.2 Energy system modelling

Energy system models are the mathematical representation of an energy system. They are translated into computer models in order to analyse them. Energy system modelling started in the 1960s. Back then, the models were built to analyse the power sector for industrial purposes [5]. Later, policy took an interest in this topic, which led to more sophisticated models on a larger scale. Because of the rising environmental concerns, a fundamental shift in the structure of energy systems and the requirements for the modelling tools had to be made. [6]

To realize the mitigation goals agreed on in the Paris agreement and to respond to safety

concerns, energy systems are changing from having centralized fossil or nuclear power plants to having many decentralized renewable energy sources. These changes come with many new challenges. In particular, the higher complexity of energy systems requires the modelling tools to be more sophisticated as well.

Roughly speaking, there are four different approaches to build an energy modelling tool [6]. One is to focus on a small scale energy system with high detail. Tools developed for this purpose can be used for example to analyse the connection of a wind turbine unite and its effect on a electrical grid. It is then possible to gain knowledge about the power flow and the stability of this particular grid. It usually operates on small time and geographic scales. Those tools are often refereed to as Power System Analysis Tools.

The second approach is similar to the first one. The tools here operate on a larger geographic scale (e.g. national level) but still on a rather short time scale. They are mostly used as a support for making operational decisions.

The third one focuses on the energy system in general and how to optimize it and transform it with new components in the future. This is the approach this thesis focuses on.

The fourth one is to examine how new policies impact the energy supply sector. An example of such a policy is the carbon tax. The third and fourth approach operate on larger time scales that can last up to several decades. Simulating and optimizing that much data is very costly in terms of computational resources.

2.3 Tessif

Tessif short for “Transforming Energy Supply System Framework” is written in the programming language Python. It aims to be an interface for different free open source energy modelling tools. Its main objective is to make the use of energy modelling tools easier for people who are not too familiar with programming in general. Furthermore, it offers a platform on which these tools can be compared and analysed. Implemented in Tessif are several open source modelling tools, for example Oemof or PyPSA. These tools were built by different organisations and hence differ in a lot of aspects. To lower the threshold of using them, Tessif generalizes several aspects of those tools like data in- or output, by transforming them into a common data format. Moreover, Tessif also comes with several visualization tools, which are very helpful for interpreting the results. In this context, one particular advantage is that the representation is independent from the chosen modelling tool. A detailed documentation is provided in [7].

Tessif interprets an energy system as a graph which contains all relevant information. This graph consists of nodes and edges, see figure 2.2. The edges usually represent the energy flow, but in theory it can be a flow of any kind. The nodes usually represent energy sources like power plants, storages, or other energy system components. Tessif can be used for a wide range of applications, from small scale modelling (e.g. the energy supply of one household) to the whole integrated European Network. However, it is optimized for medium to large scale energy systems. Implemented in Tessif are several functions to analyse optimization and modelling results of the integrated energy system modelling tools. They help with the general understanding of the results and can also provide information about which tool is best suited for the given task. Next to the accuracy and

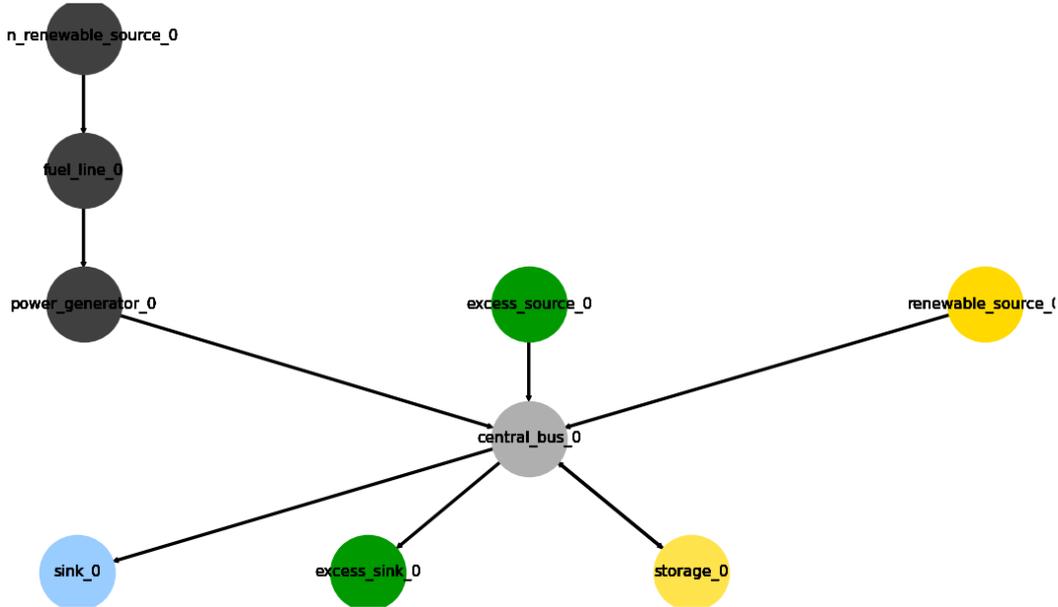


Figure 2.2: Representation of an Energy system as a graph(own figure generated with Tessif)

reliability of a tool it is also important to keep an eye on the computational resources needed to generate the results. A large scale simulation like the previously mentioned optimization of the integrated European Network is a very complex task which needs a lot of memory space and time. This corresponds to high costs for the end user. In this thesis, new ways to measure the amount of computational resources are implemented and reviewed.

2.4 Oemof

Oemof was founded in 2014 by researchers from the Center for Sustainable Energy Systems in Flensburg in cooperation with researchers from the Reiner Lemoine Institute in Berlin. It was originally developed to model energy systems from the heat and power sector, but as time passed more researchers from other facilities joined and the Oemof modelling framework grew bigger. New packages were included which specialize on different tasks of the energy system modelling process. The program and documentation can be found in [8]

The most famous Oemof package is oemof.solph which is also the package used in this thesis. It is commonly used to generate tools for either mixed linear optimization or linear optimization of energy systems.

The oemof.solph package builds a graph consisting of busses and energy system components very similar to Tessif. These busses and components are connected via edges. They represent the direction of e.g. the energy flow or the direction of fossil fuels. This graph contains all the information about the energy system and the interactions of different components with one another. The graph will then be transformed into an optimisation model. The optimisation process will be done automatically by the python based open

source software package “pyomo”. This simplifies the process substantially, because an in depth knowledge of the mathematical background is not strictly necessary, allowing for usage by a wider range of people.

The energy system is optimized and solved with respect to several parameters. The main goal is to minimize costs. These costs do not have to be exclusively of economic nature. They can also be of environmental or technical nature. Finally, it is also possible to simulate different investment scenarios to help make decisions on which form of energy production to invest in.

2.5 PyPSA

PyPSA short for Python for Power System Analysis is an open source toolbox written in the programming language Python. It was developed at the Frankfurt Institute for Advanced Studies and is currently maintained by the Department of Digital Transformation in Energy Systems at the Technical University of Berlin. It is financed by the German Federal Ministry for Education and Research and its goal was to do simulations for the CoNDyNet project. The objective of that project was to analyse future electrical grids. The focus lays on stability, risks and economic performance.

PyPSA is commonly used to analyse modern electrical grids with both fluctuating renewable energies and constant energy supplying fossil fuels. It is optimized for large networks and timescales. The principle way of working of PyPSA is quite similar to that from oemof. Again, there are several components which can be put together to an energy system. As in oemof the optimization is done with the software ‘pyomo’. A detailed documentation can be found in [9]

2.6 Computational complexity theory

The computational complexity theory is a sub-field of theoretical computer science. It focuses on the resource usage of computational problems. In order to analyse a problem, it has to be computable or effectively decidable. This means it is solvable by a computing device (i.e a computer.). In the context of the classical computability theory every effectively decidable problem has the same difficulty, namely that it is solvable. However, there can be huge differences in the practical difficulty. We will use methods from this theory to analyse our modelling tools. [10]

2.7 Computational complexity

The objective of computational complexity is to analyse an algorithm regarding its efficiency. The efficiency of an algorithm correlates with how much computational resources are needed to execute it. Usually the time and memory space consumption are considered, but there are also other interesting parameters like electrical power or computer cores. In this thesis, we will restrict ourselves to time and memory space consumption. Time and space complexity is mostly expressed using Bachmann-Landau symbols which are introduced in Section 2.22.

2.8 Time complexity

Time complexity is an important part of computational complexity. It describes the amount of time an algorithm needs to run. It can be determined by measuring the run time of the algorithm, but more commonly there is a different approach to measure the time complexity. With computers getting faster and faster the measurement of the run time loses its reproducibility. An algorithm which took a few minutes to run in 1970 takes now maybe about a few milliseconds. There are even big differences between current computing devices. A better approach is to measure the amount of operations the algorithm needs. The number of operations did not change from 1970 to now if it is still the same algorithm. The approach to measure the amount of steps taken by an algorithm might be a good idea for smaller algorithm. However in the context of Tessif, which is simulating big and complex energy system through tools like Oemof and PyPsa, it is hard to count the amount of operations. Even with these open source tools, which means that everyone has an insight in the source code, it is too complicated to retrace and count every step the algorithm took. Keeping that in mind it is much easier to measure the time needed by the algorithm. Furthermore Tessif aims to be a platform for not only computer scientist. For engineers it is much more interesting to know how long the algorithm will need on their specific computer than to know how many operations are taking place.

2.9 Space complexity

The space complexity is very similar to the time complexity. It is also a part of the computational complexity and describes the memory requirements of an algorithm. It is determined by measuring the needed memory space.

2.10 Scalability

Scalability describes how a system is able to deal with a growing amount of work. The most interesting question is how the resource consumption behaves with a growing input size. Again, the scalability is measured in terms of Bachmann-Landau symbols. To measure the scalability of an algorithm a duplication algorithm is often used, however there are different options available in computer science. This thesis uses a form of the duplication algorithm, which means that smaller, in this case, energy systems can be connected to a larger one. Additionally to extending the scale of the energy system it is also possible to vary the timescale.

2.11 Python

As already mentioned, the energy system modelling framework Tessif, as well as the implemented modelling tools are written in the programming language Python. Python was created in the early 1990's by Guido van Rossum in the Netherlands. In 2001 the Python Software Foundation (PSF) was formed, mainly for intellectual content related

to this programming language. PSF is a non profit organisation and with this Python became an open source software and still is today [11].

Open source means basically that the software and the source code is accessible for everyone. Also changes and modifications must be allowed. The idea behind Python was to create an easy object oriented language with a simple but strong core, which can be easily extended even by external programmers. They can create modules and libraries, which can be implemented via the “import” statement. In the following, some of these modules are discussed, which are used or were considered to be used for analysing the complexity of our algorithms.

2.12 Time and space measurement

There are various methods in the Python programming language universe to measure the amount of time or memory space needed to execute algorithms. Most of them work through inbuilt libraries or modules and are specialized on different cases, e.g. small code snippets or large functions. In the next chapter some of them are introduced and we will justify our decision of the chosen module [11].

2.13 Time libraries

2.13.1 Time library

The time function has many uses. It can be used in a function to state the momentary time, day or year but it is also possible to measure the time with it. With the function “time()” of the time module it is possible to return the passed time since the epoch. The epoch is the point where the time started and can vary from platform to platform. For Unix products this is January 1, 1970, 00:00:00 (UTC). This alone can not give any information about how much time a specific task needs but if this function is executed before and after running the algorithm the difference can be calculated. This provides an idea of how long the given task takes. This is a bit too inaccurate for the purpose of this thesis. But there are other inbuilt functions which are more accurate. One example is the “process_time()” function. This function returns the sum of the users CPU time and the time of the system needed for the current process. This also works process wide. To get an accurate result it is necessary to measure the time before and after the execution of the algorithm. After this, it is possible to calculate the difference and there is a valid information about the needed time [12].

2.13.2 Timeit library

The python module “timeit” was designed to analyse small snippets of code. It executes the snippet per default one million times and calculates the average time consumption [13]. To measure the time the timeit module uses the function “process_time” from the python library “time”. However, this module is created for small snippets of code and therefore not ideal to use in this thesis. Indeed, with algorithm taking up to some minutes the default setting of one million repetitions would take way too long.

2.13.3 Decision on the time measurement library

In view of the arguments outlined above, we decided to use the time module with the “process_time” function. This method has a low threshold, is easy to understand and is sufficiently accurate for this thesis. Also in comparison to the timeit function it can be better implemented into functions.

2.14 Memory measurement libraries

The next subsection outlines possible ways to analyse the amount of memory space needed by an algorithm. To this end, different libraries are discussed. Finally, we justify our decision on which one to use.

2.14.1 tracemalloc

The library “tracemalloc” [14] was designed to be a debugging tool to find memory leaks by tracing memory blocks allocated by Python. To analyse possible memory leaks there are several functions in the module which are also able trace the memory usage. The function `tracemalloc.start()` starts the analysing process and the function `.get_traced_memory()` returns the size of the the memory blocks traced by the “tracemalloc” module. Additionally, it returns the peak sizes of those allocated blocks. Both are returned as integers in a tuple. Of particular interest for this thesis is especially the first integer, which is the traced size of memory space. The `.stop()` function clears all the previous collected data by the module.

2.14.2 psutil

“psutil” [15] is a library for Python and is mainly designed to analyse running processes. It can be used to monitor and profile systems. However it is primarily designed for actual system monitoring and not for tracing the memory usage of a specific function implemented in a program. Therefore, it is not a very good fit for the task at hand.

2.14.3 Decision on memory library

After an thorough examination of the possible modules to analyze the memory space usage the decision fell on the tracemalloc library. Even if it is designed to be a debugging tool it still provides the lowest threshold of using.

2.15 Implementing functions into Tessif

The following chapter explains the implementation of the previous introduced theoretical basics in the context of Tessif. The goal is to provide a low threshold analysing tool for computational complexity questions. To this end, new functions and algorithms are build. In addition, we also provide ways of visualizing the results. The complexity

functions will be implemented into pre-existing algorithms, but also new energy systems will be constructed.

Before explaining these new functions, this thesis will provide an thorough explanation how the modelling framework Tessif works.

As mentioned before, Tessif aims to be a low threshold modelling platform. It should be possible to use different modelling tools through it. Therefore, Tessif supports various data formats which sometimes also extends the range of the other integrated modelling tools. For example, through Tessif it is easier for Oemof to use data formats Oemof does not support. How this is possible is described in more detail in the subsequent paragraphs.

Tessif supports several data formats like for example .hdf5, .cfg, .xml, or even Python files. This data formats can be transformed via the parse module. With the help of this module the information stored in the files are transformed into a Python mapping.

The next step is transforming this mapping into the Tessif specific energy system, using the transform module.

This Tessif energy system can now be transformed into any energy system associated to one of the various implemented modelling tools. For example, if we want to use Oemof for the optimization than Tessif will transform it into an Oemof energy system.

The next step is the actual optimisation. The energy system will be optimized regarding the chosen parameters and modelling tool.

The last step is called “post-processing“. The optimized energy system is transformed back into a Python mapping. This way the results can always be evaluated and visualized in the same way independently of the modelling tool. In the following, an algorithm will be introduced which measures the resource requirements of each of these steps.

2.16 Time measurement

The time measurement function has two objectives. The first one is to provide a tool which can analyse the time consumption of one specific energy system simulation process. The other is to provide a working bases for the scalability function. This function will call the `stop_time()` function and iterate through energy systems of different sizes and timescales to see how the resource requirements change with growing Input.

The time of each of the previous mentioned steps is measured. The “time” module with the `process_time()` function was chosen to do this. The `process_time()` function returns the system and CPU time of the current process. In order to get the time of a specific process the function must be called before and after the algorithm. The difference of this two function calls is the amount of time the computing device needs to execute the algorithm. However, this is not the real time because the computer also makes other things simultaneously which are not taken into account by this time module function.

The first float integer which the `process_time()` function returns will be stored in the variable `start_time1`. After that the first of the above outlined steps takes place. The information are extracted from the external data format and turned into a Python mapping. This mapping will be stored in energy system mapping. In Python each

line of code will be executed consecutively. Thus, when the energy system mapping is built the next line of code is executed. The `process_time` function is called again and stored in the variable `end_time`. The actual time which the process needed is now the difference between the two measured times and is stored in the variable `reading_time`. The last line of code rounds the time after the fourth decimal place and stores the result in the dictionary “`timing_results`”. This method will be repeated for the other four other steps.

2.17 Memory space measurement function

The algorithm to measure the memory usage is very similar to the time algorithm. We briefly outline it here again. As previously mentioned the module `tracemalloc` was chosen. The `start` function starts the tracking process.

After starting the process the first step of the energy system modulation process takes place. The external data will be turned into an energy system mapping. After this step the `get_traced_memory` function returns a tuple with two integers. The first integer represents the amount of memory space which were traced by the `tracemalloc` module. This result is stored into the variable `reading_memory` and then the `stop` function will delete all the allocated memory blocks of the module. This is necessary, because otherwise there could be an inaccurate result if the function is used again. This method is repeated for all the subsequent steps.

2.18 Functions to give a quick overview of Energy Systems simulation

In the following sections we discuss the functions we built to analyse the resource requirements of an energy system optimization in terms of time consumption (stop time), memory consumption (trace memory) and scalability assessment (asses scalability). Moreover, we also built visualization functions, which are part of the discussion as well. The construction of the measurement functions will essentially follow the strategy outlined in the last two sections.

One objective is to make these function as variable as possible, meaning that it is possible to simulate all energy systems data formats and with all energy modelling tools that Tessif supports.

2.18.1 stop time function

The function we built to analyse the time consumption of the energy system simulation in Tessif is the `stop_time` function. We will outline its main characteristics below. This function has four input parameters. The path, the parser, the simulation tool and the timeframe.

The function simulates the whole process of simulating an energy system through Tessif. From parsing the external data to the post-processing. For each of those steps the time is measured.

The first step is to extract the data from the external source and transform it into a Python mapping. The path points the parsing function to the location where the data is stored. Depending on the data file a different parser is necessary which can be specified as an input parameter. Before and after extracting the data the `process_time` function is executed. The difference, which is the time the processor needed for that code execution, is measured. After parsing the data, it will be transformed into a Tessif energy system. This energy system can now be transformed into an energy system specific to all the modeling tools implemented in Tessif. All these steps are happening in the Tessif framework. Now the actual optimization takes place. The energy system is simulated with the chosen modeling tool. The modeling tool is specified with the second parameter. After the optimization the data is transformed back in the post-processing into a mapping. For each of those steps the process described in the beginning for the creation of the mapping is done again.

All the results are stored into a dictionary keyed to the notation of the simulation steps.

2.18.2 trace memory function

The measuring of the memory space is very similar to the time measurement. The input parameters are the same as well as the measured steps. The difference lies in the external python library. The structure of the function is the same as for the `stop_time` function.

Before each of the modeling steps the `tracemalloc` function `tracemalloc.start()` is used. After each step the `get_traced_memory()` extracts all the traced memory up until that point. After that the `tracemalloc.stop()` function is used which stops the analysing process and clears all the previous data regarding the memory measurement. That is necessary because otherwise time memory blocks would supersition for every step.

The function to measure the time of a modeling process is called `stop_time` and for the memory space, `trace_memory`. Both of this two function have four parameters to define them. The first one is the path to the location of the file, which contains the energy system information. The second is to specify the parser. Depending on the data format of the file a different parser is necessary to read the file and extract the data. The third parameter defines the modelling tool and the last one the timescale. The timescale is set on a specific one per default but can be changed to the liking of the user. Both functions return a dictionary, where each measurement is keyed to the belonging step. Building the functions like this makes it easy to use them on the one hand for quick measurements of energy systems but also to use and call them in more complex functions. This will happen later in the scalability function.

2.19 Visualization of stop time and trace memory

For better understanding we also implemented a visualization function. This function is able to call the previous mentioned functions `stop_time` and `trace_memory` and builds a instructive plot from the returned dictionary.

Figure 2.3. shows an exemplary simulation process with Oemof of the fully parameterized working example, which is included in the Tessif example repository. It showcases how

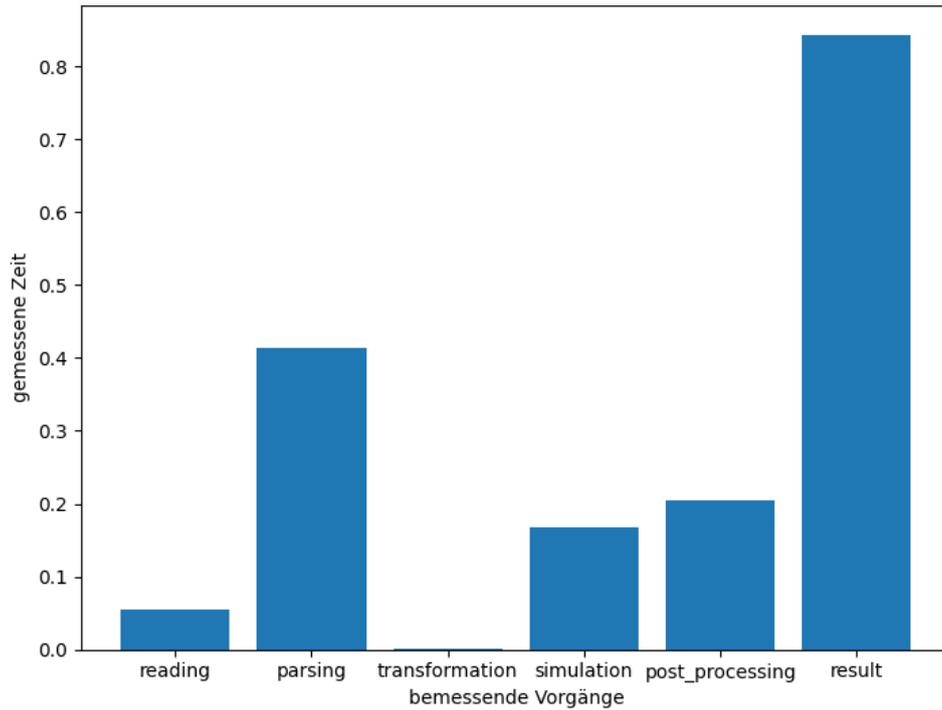


Figure 2.3: bar plot of the time visualization function (own figure created with Tessif)

long each step took to execute as well as how long the whole process was. The plot for the `trace_memory` function has the same structure.

2.20 Scalability

The goal of the function described subsequently is to provide a tool, which analyses the scalability of an algorithm. As mentioned above, scalability is in general a parameter to describe how a system copes with a growing amount of resources. In terms of computer science the scalability describes how the algorithm or program deals with a growing input. The function analyses the scalability of energy system modelling tools in Tessif regarding their computational complexity. In order to realize that a duplication algorithm is used. For that algorithm we designed a small energy system.

2.20.1 Creation of a minimum self similar energy system

This energy system consists of three energy sources. One symbolizes a renewable source, for example a photovoltaic module. Another one generates energy through fossil fuels. Both sources create a randomized energy output which is unique for every simulation attempt. The last source serves as an excess source and is needed for the solvability of the optimisation process. The excess source and the renewable energy source are both connected to the main bus. The fossil fuel source is connected to a transformer which transforms the fossil fuel into energy. The transformer is also connected to the main

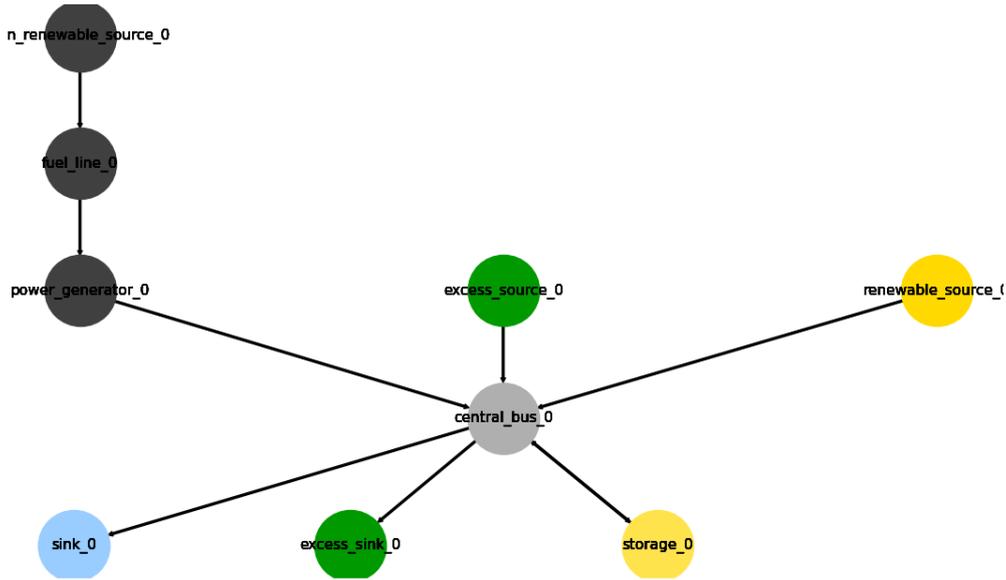


Figure 2.4: minimum self similar energy system (own figure created with Tessif)

bus. The bus is the main part of the energy system and represents a simplified version of the electrical grid. It connects most of the energy system components to each other. A storage unit and the sinks branches off from this main bus. The storage is able to store excess energy from one time-step to the next. A time-step can represent a day or a year depending on the chosen model and own specification. One sink represents the normal energy demand and the other one is an excess sink. It works similar as the excess source and is needed for the solvability. The last component included in the function is the connector which is only needed when two or more minimum self similar energy systems are connected with the create self similar function.

2.20.2 Creation of a self similar energy system

The 'create self similar energy system function' is the foundation of the scalability analysis. In this function an adjustable amount of minimum self similar energy systems can be connected to a larger one. Therefore energy systems with growing scale can be analysed.

With the help of a connector it is possible to connect the two main busses of two minimum self similar energy system and create a larger system. The energy system will be called minimum self similar energy system and is an overly simplified representation of an actual energy system. But in the context of scalability it enables the collection of information.

The 'create self similar energy system function' is defined by two input parameters. The amount of minimum self similar energy system that should be connected and the timeframe.

The timeframe consists of the information of how many timesteps, in this case days, should be simulated. The function then connects the number, specified in the parameter by the user, of minimum self similar energy systems to one big energy system and also

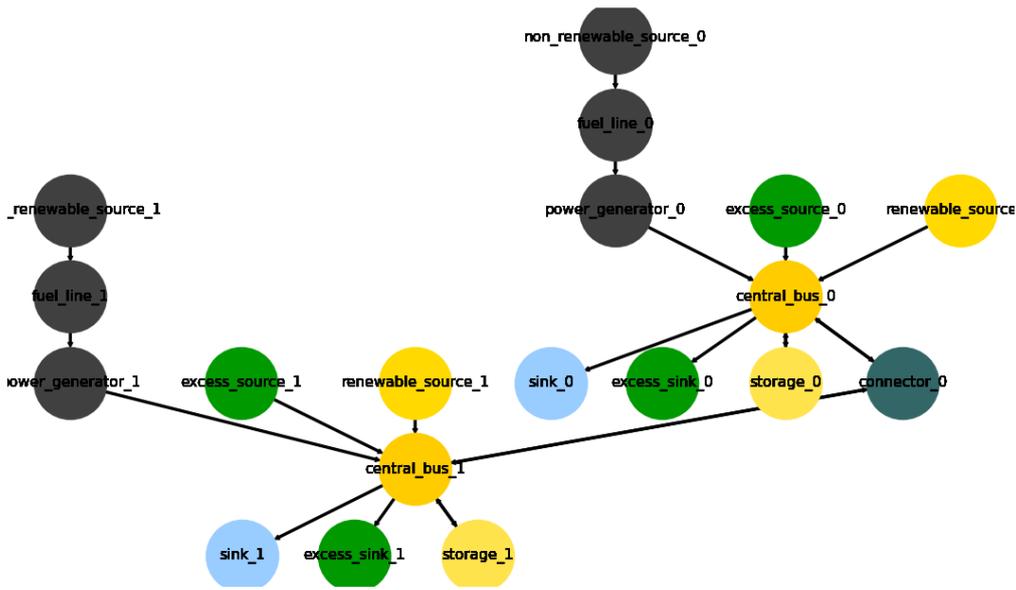


Figure 2.5: Two minimum self similar energy system connected into a self similar energy system (own figure created with Tessif)

stores the information of how many timesteps should be simulated. This big energy system is then ready to be simulated.

2.20.3 asses scalability

The function to measure the scalability is called 'asses_scalability'. The algorithm calls most of the previously explained functions. There are four input parameters to specify: the amount of minimum self similar energy systems that should be connected in the end, the timeframe, the storage location and the modeling tool.

As shown in the 'flow chart' (see Figure 2.6) the first step of the function is to generate the timeframes. The input parameter for the timeframe is called 'T' and is an integer. The dataformat for the timeframe on the other hand is a list. This list contains the amount of days that are simulated in the function. Therefore the integer is transformed into the required data format as well. In the end four different timeframes are stored in that list with different amount of days. The integer T represents the largest timeframe, where the amount of days equals the value of T. The other three timeframes are evenly distributed from T to zero.

After having specified the timeframe the first 'for loop' starts. The loop uses the input parameter N, which represents the amount of minimum self similar energy systems. It iterates N-times through the loop connecting each time another minimum self similar energy systems to the already existing energy system.

Inside that first 'for loop' is a second loop, which iterates four times through the timeframes, once for every timeframe.

In this second loop energy systems are generated, stored on the computing device and analysed. The first step is to create the energy system. This is done by the `create_self_similar_energy_system` function, which was described earlier. This function uses the current value for N and the timeframe as input parameters and creates the energy system. Next this energy system is saved at the storage location. The data format is an hdf5 file. After storing the energy system at the specific location, it is called by the `stop_time` and the `trace_memory` function. The processor time and the memory usage of that simulation is measured and stored in a list. The results are collected for every iteration. Therefore $N \cdot 4$ energy systems are simulated which makes it possible to analyse how the system copes with a growing input size. In other words, we get an impression of the scalability.

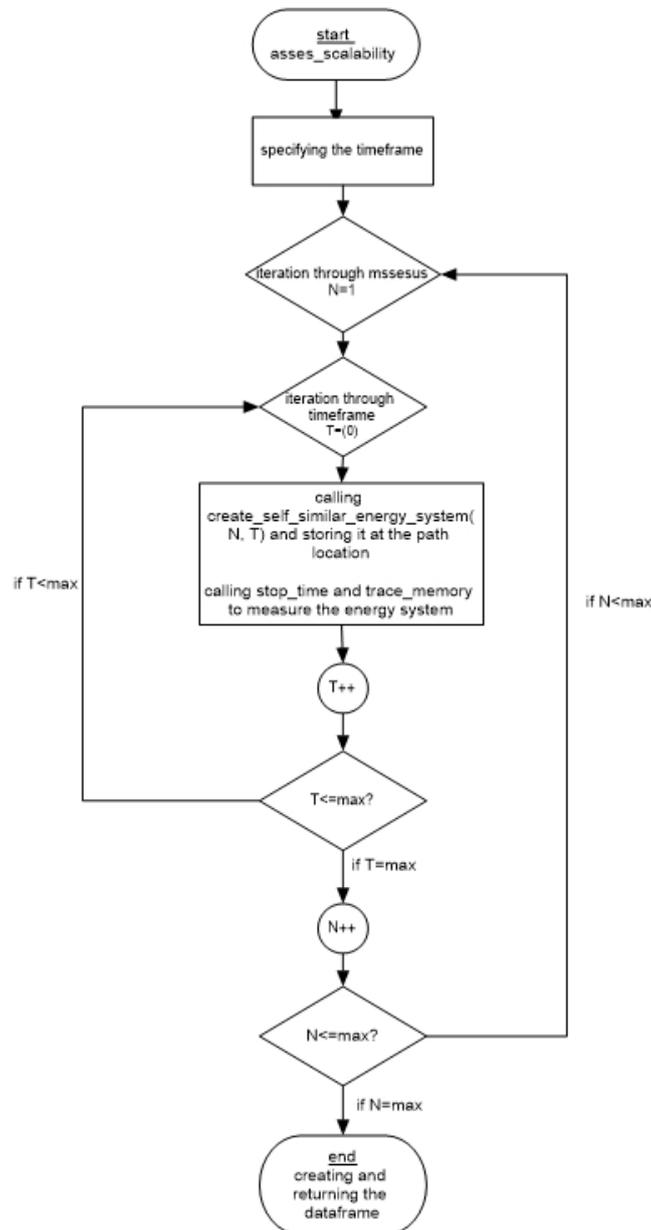


Figure 2.6: Flow chart of the `asses_scalability` function (own figure)

2.21 Visualization of scalability functions

We created two visualization functions for an easier understanding and a better overview of the results. The first one provides a general overview of the growth of the total resource requirements. The result are presented in a 2D line plot. And the second one takes the specific steps into account. For each energy system a stacked bar plot will be created.

2.21.1 2D visualization

The 2D visualization function is called `visualize2D_asses_scalability`. The function calls the `asses_scalability` function and works with the returned data frames. The time and memory space data is stored individually for every step of the simulation. The motivation of the 2D visualization function is to obtain a general idea of the scalability of energy system simulation in the Tessif framework. Therefore the total amount of computational resources is calculated by adding the individual steps.

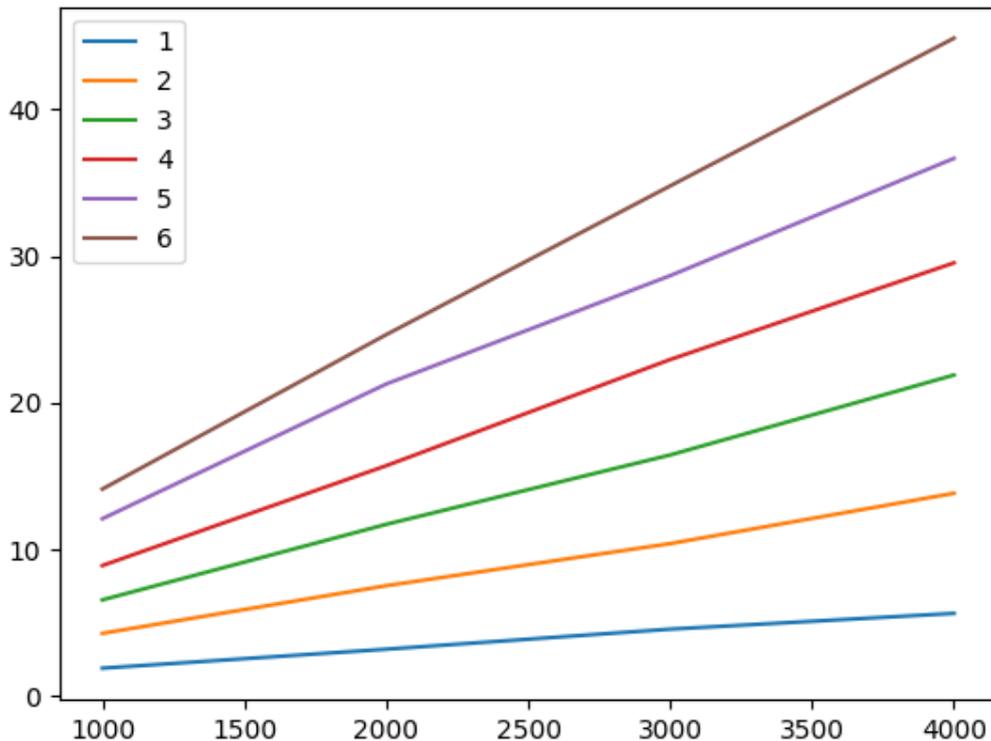


Figure 2.7: 2D visualization of the energy systems optimized with oemof

Figure 2.7 shows how the time consumption behaves for energy systems of different sizes and timescales. Each of the lines represent how the time consumption of an N-msses energy system grows after enlarging the number of time-steps. Additionally it is possible to compare the elapsed wall time of different sized energy systems at the same timescale. The different period number examined in the graph were 1000, 2000, 3000 and 4000 and the number of mssesus went from one to six.

As mentioned above, this graph is used to get an idea of the scalability of the modelling tools simulated through Tessif. It provides limited information on the scalability of the

modelling tools itself, because steps were taken into account which are not necessary for simulating the energy system directly with the modelling tool. In order to get an idea about that, we created a second function. This differentiates more between the individual steps and will be outlined in the next section.

2.21.2 3D visualization

Memory Usage Results of Model: 'oemof'

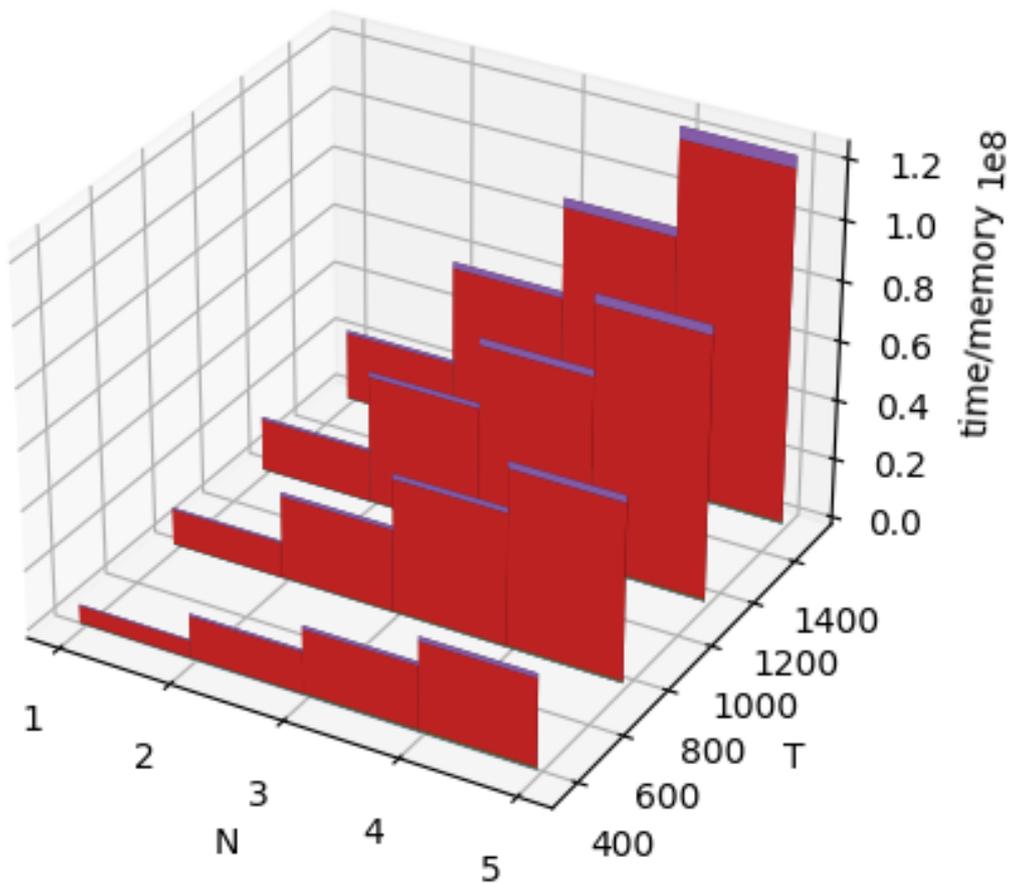


Figure 2.8: Stacked 3D bar plot of a simulation with Oemof

Each of the colours in Figure 2.8 represent one of the step previously outlined above and in the graphic for the `stop_time` function. The size of the energy systems ranged from one too four msses. The timescale went from 100 to 400 in steps of 100. Each stacked bar plot presents how much time was needed for the individual step and the height of the stacked bar shows the total amount of time for the whole process. The graph for the memory usage is similar to the shown one but presents the other belonging information. However, the limits of this graph appeared when we were simulating larger energy systems on a bigger timescale. Some of the 'steps' are nearly independent of

the size and timescale of an energy system and can therefore become ‘invisible’ when visualizing bigger energy systems. (steps mean in this context and in general in this thesis the mentioned steps in the `stop_time` and `trace_memory` function. simulation and post processing grow linear but the transformation or the generation of the mapping is independent from the size of the energy system).

2.22 Bachmann-Landau symbols

The big-oh notation (\mathcal{O}) also called Bachmann-Landau notation or asymptotic notation is a mathematical notation, which describes the asymptotic behaviour of a function. In computer science it is mainly used to describe the complexity of an algorithm and is therefore a tool to analyse the computational resources of computer programs. The formal definition is as follows [16]:

Definition 1 (Bachmann-Landau notation). *Let $f : \mathbb{N} \rightarrow \mathbb{R}_+$ and $g : \mathbb{N} \rightarrow \mathbb{R}_+$ be positive functions, both defined on the natural numbers. Then we write*

$$f \in \mathcal{O}(g)$$

if there exists $N_0 \in \mathbb{N}$ and a constant $C \geq 0$ such that

$$f(n) \leq Cg(n) \text{ for all } n \geq N_0.$$

In addition, we write

$$f \in o(g) \text{ if } \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = 0$$

In our setting, the variable n usually represents the input size and the function $f(n)$ the time used to compile the algorithm. The function $g(n)$ is then used to give an idea of the complexity of the algorithm in standardized terms. For example, commonly used classes include

1. $g(n) = \log(n)$ meaning that the complexity is at most logarithmic
2. $g(n) = n$ meaning that the complexity is at most linear
3. $g(n) = n^2$ meaning that the complexity is at most quadratic

Finally, constant factors and small input sizes are in this context irrelevant, which can be seen directly from the definition.

3 Evaluation

In this chapter we evaluate the computational complexity of the two modelling tools Oemof and PyPsa through Tessif. This is done with the help of our built functions, which were documented in Chapter 2.

To this end, we start with the verification of the previously built functions. We introduce a suitable measure of dispersion and verify the statistical correctness of the functions.

Then we look at two different scenarios: In the first we treat the timescale as a variable and analyze how a growing timescale influences the necessary computational resources (time and memory). In the second scenario, we treat the size of the system as a variable and analyze how a growing size influences the necessary computational resources (time and memory)

3.1 Verification of the collected data:

3.1.1 Measure of dispersion

Each simulation process generates different results. To get an idea of the scale of their dispersion, we will conduct several simulations and analyse the obtained results.

There are two commonly used sets of parameters to analyse a set of data [17]. In this context, a set of data consists of the collected results (called characteristics henceforth) of the simulation process, for example the amount of time or memory space each iteration needed to compute.

The first set are the so called location measures. Examples of location measures are the median and the mean average. However, these parameter do not give any idea about the general dispersion of the characteristics. Indeed, two sets of data can have the same mean average even though their characteristics differ drastically. For example, consider the two datasets [5,7] and [3,9], which have the same mean average (six).

The second set of parameters are the so called measures of dispersion. Their purpose is to provide more detailed information about the the dispersion of the characteristics. One example of a measure of dispersion, which is easy to calculate is the range of the values. The range is calculated by the difference of the biggest and smallest characteristic. However, this is very sensitive with respect to statistical outliers and can therefore be rather misleading. Another, and more promising, parameter is the variance (together with the standard deviation). The variance is a quadratic dispersion measurement, which has the disadvantage that it is less informative at the first glance than the range. Because of that one often considers the standard variation, which is the square root of the variance. The variance (and the the standard deviation) provide information about

the distance of the characteristics to the average mean. We are now giving the formal definitions of the above introduced terms.

Definition 2 (Standard variation). *Let $\{x_i\}_{i \in \{1, \dots, n\}}$ be a set of data, where n is the total amount of characteristics and x_i the value of each characteristic. We define the arithmetic mean as*

$$\bar{x} := \frac{1}{n} \sum_{i=1}^n x_i$$

and the variance as

$$\begin{aligned} s^2 &:= \frac{1}{n} [(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2] \\ &= \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \end{aligned}$$

Finally, the standard deviation is defined as the square root of the variance, i.e.

$$s := \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$$

Definition 3 (range). *Let $\{x_i\}_{i \in \{1, \dots, n\}}$ be a set of data, where n is the total amount of characteristics and x_i the value of each characteristic. We define the range as*

$$R := \max_{i \in \{1, \dots, n\}} x_i - \min_{j \in \{1, \dots, n\}} x_j$$

We will calculate the variance of our collected data sets. After the pre-evaluation of the data we expect that the standard deviation is bigger for smaller energy systems and becomes smaller as it grows. It also became clear that the first iteration of each simulation process takes up more time than the following. One possible reason for this is that certain data is generated in this first step which is stored and used in the following iterations (and therefore does not have to be generated again). Because of that, we decided to not include the first simulation for the calculation of the variance and standard deviation.

As we can see from Definition 2, the first step is to calculate the arithmetic mean. Then the arithmetic mean will be subtracted from each characteristic and the result squared. Finally, we add things up and divide by the total number of characteristics.

3.1.2 Calculation of the measure of dispersion

In order to analyse the statistical quality of the implemented tools, we calculate, as mentioned above, the standard deviation. To this end an energy system consisting of 5 minimum self similar energy systems and 4000 timesteps is simulated several times and the computation time is measured. Then we determine the arithmetic mean, the variance and finally the standard variation. The results are summarized in the following figure.

	reading	parsing	transformation	simulation	post_processing	result
	0.94	1.17	0.01	31.4	4.38	37.9
	0.6	0.73	0.01	31.46	4.37	37.17
	0.61	0.75	0.01	31.41	4.32	37.1
	0.67	0.87	0.01	31.41	4.84	37.8
	0.66	0.8	0.01	31.43	4.41	37.31
	0.6	0.77	0.01	31.28	4.6	37.26
	0.73	0.73	0.01	31.53	4.39	37.39
	0.74	0.79	0.01	31.46	4.53	37.53
	0.72	0.75	0.01	31.33	4.69	37.5
	0.78	0.75	0.01	32.24	4.45	38.23
arithmetic mean	0.705	0.811	0.01	31.495	4.498	37.519
variance	0.009725	0.015889	0	0.065985	0.024896	0.115489
standard variation	0.098615415	0.126051577	0	0.256875456	0.157784663387 >	0.339836726

Figure 3.1: calculation of the standard variation (own figure)

The above described energy system was simulated ten times. We can see from Figure 3.1 that the first simulation was in terms of 'reading' and 'parsing' slightly off. As already mentioned, this is presumably due to the way data is stored internally in the computer. Indeed, the data is stored in the primary memory and can be directly called from there again. The measured times are rounded to a hundred of a second. The standard deviation is quite small already. This means that the functions are working well for medium to large sized energy systems.

We also examined energy systems of different sizes, and realized that it seems to be the case that the larger the energy system is the lesser is the variance. In particular, the dispersion grows substantially for very small sized energy systems.

3.2 Analysis

In the following section the resource requirements of energy systems modelled by the energy system modelling tools Oemof and PyPsa are analysed. In the previous chapter we introduced various functions, that will be tested now for their reliability to analyse small energy system as well larger ones. After reviewing the integrity of the self similar energy system (see Subsection 2.20.2) the first measurements will take place.

We start by determining the complexity class for the individual modelling tools as well as for every step of the simulation process. After that there will be an comparison of the two modelling tools Oemof and PyPSA and we give possible explanations for the existent differences. Additionally to the scalability of the systems, it will be also discussed how the resource requirements vary from small to big energy systems and with statistic verification how reliable the information are.

The computer used for the analysis runs with a 1.6 Ghz Dual-Core processor and 16 GB RAM on the operating system Linux with Ubuntu 20.04.

3.3 Scalability regarding the timescale

In this section we review the behaviour of the system if the timescale is enhanced. To this end, the `asses_scalability` function is used. For the moment, we are only interested in how the timescale influences the scalability. Therefore, our energy system consists of only one minimum self similar energy system. The scale of the timeframe increases throughout the simulation.

In the beginning, the process time is analysed. To this end, one energy system was simulated ten times, each time with a different timescale. The scale went up in equal steps from 5000 to 50.000. This was done for both integrated modeling tools

3.3.1 Expectation

Five different simulation steps are reviewed by the `stop_time` and `trace_memory` function. For each of those steps the complexity class is determined. Recall that the first step is parsing the external data into a python mapping. The second one is the transformation into an Tessif energy system. The third one is the transformation into the model specific system, the fourth one the optimization and the last one the post processing.

We conjecture that the first two steps need the same time to execute no matter which model is used. The extraction of the data and the transformation into an Tessif energy system are completely independent from the modeling tool, therefore the computational resources should be the same no matter which tool is used later.

On the other hand, we expect the resource consumption of the transformation from the Tessif energy system to the energy system of the modelling tool to differ. Tessif and Oemof use similar interpretation of energy systems, which is why the transformation should be quicker with Oemof than with PyPSA. However, we are not sure of how big this difference will be in the end and in particular if their complexity class will coincide or differ.

It follows the optimization step. Recall that PyPSA and Oemof both use the solver 'pyomo' (see section 2.5). Therefore, we conjecture that both tools have the same complexity class. However, the actual resource consumption may still vary, because of the different interpretation of energy system that are solved by the same software.

In the post-processing the optimized energy system is transformed back into a python mapping. Because both modeling tools use the same solver we predict that they need similar computational resources to execute and have the same complexity class.

3.3.2 Oemof time complexity

As specified in 3.3 the energy system was simulated ten times with different timescales. Figure 3.2 shows the simulations and their outcome.

We see from Figure 3.2 that the growth of the computational resource time is linear with respect to the timeframes. Consequently, in terms of the introduced Bachmann-Landau symbols (see Section 2.22) the complexity class is $\mathcal{O}(n)$

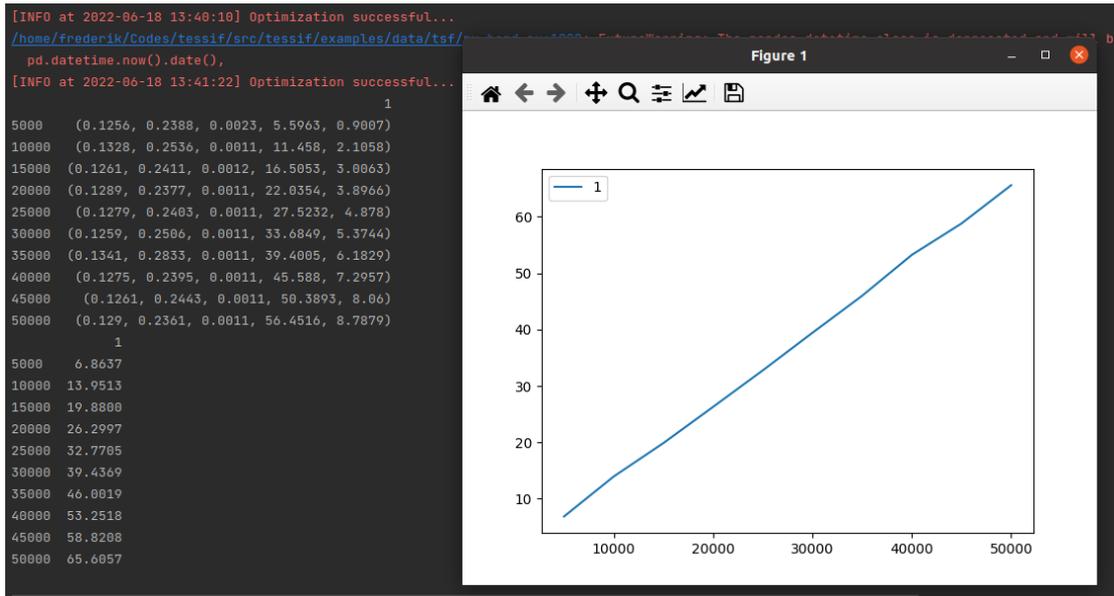


Figure 3.2: results of the Oemof time measurement (own figure created with Tessif)

We see from the rough data on the left hand side of Figure 3.2 that the timeframe makes no visible difference in the beginning. The timeframe consists of a list containing integers which represent the dates for the days. This list grows linear with respect to the timescale. However, as it has been observed it does not significantly influence the amount of time needed to extract the data. In addition, also the parsing from the external data, the transformation into an Tessif energy system and the transformation into the Oemof energy system stays the same for each timeframe. This outcome was expected, because the timeframe is a parameter specifying the amount of days simulated in the optimization process. Consequently, it is reasonable that it has little impact on the previous steps.

The simulation step however is heavily influenced by the timeframe. The first simulation step with a timeframe consisting of 5000 timesteps takes about 5,6 seconds. By doubling the timesteps to 10000 the needed process time doubles to 11,5 seconds. The last timeframe with 50.000 timesteps takes 56,5 seconds which is about ten times the amount of the first timeframe. Therefore, the time consumption grows proportionally with the timescale. The post_processing is similar to the simulation step. The required time is again proportional, but the starting point is smaller than the one for the simulations.

We remark that the simulation step is by far the most relevant step in terms of absolute time consumption. Therefore, the complexity of the entire process is dominated by the complexity of the simulation step and their complexity classes are the same.

3.3.3 PyPSA time complexity

The time analysis in PyPSA for the timescale was conducted in the same way as for Oemof. The same energy system was simulated ten times with the same timeframes as the input.

From Figure 3.3 we learn that the time consumption is once again proportional to the timeframe. Therefore, the complexity class is $\mathcal{O}(n)$ for the whole simulation process.

However, the actual time consumption is larger than for the optimization process with Oemof. By reviewing the individual simulation steps, we see that there are especially differences in the transformation from the Tessif energy system to the PyPsa energy system as well in the optimization process. We predicted differences for the transformation (see Section 3.3.1).

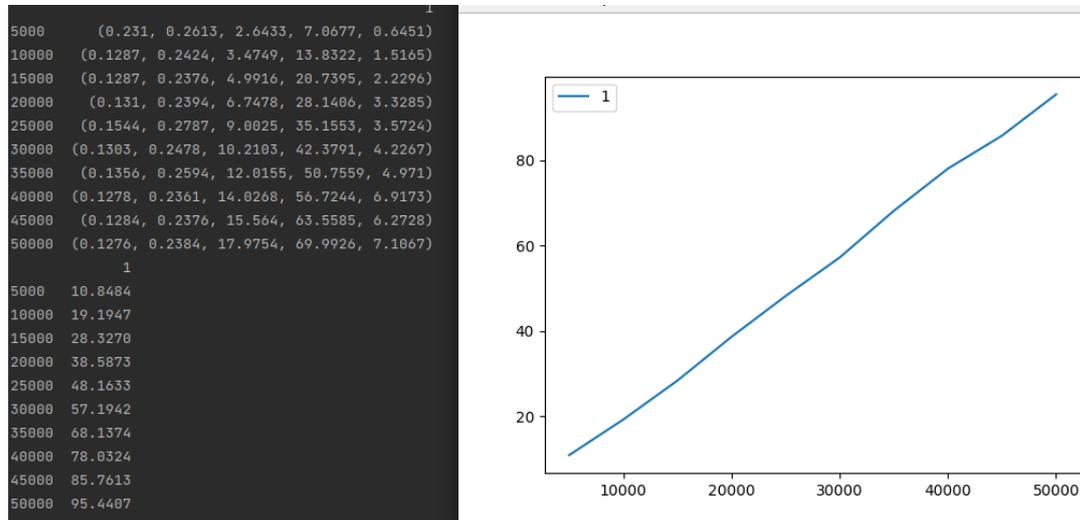


Figure 3.3: results of the PyPSA time measurement (own figure created with Tessif)

A thorough step by step analysis reveals the following: The first two steps behave very similar to Oemof, their complexity class are again $\mathcal{O}(1)$ and also their actual time consumption is comparable. The same is true for the last step, here the complexity class is $\mathcal{O}(n)$.

The third step, the transformation from the Tessif energy system to the PyPSA energy system, is the most interesting one. As we can see from Figure 3.3. the time consumption for this step is proportional to the timeframe, which means that the complexity class is $\mathcal{O}(n)$. Therefore, this step does not only take more time than with Oemof (which we expected) but it even has a different complexity class. We also observe, that this step is consistently the second biggest contributor.

The complexity class of the optimization process is again $\mathcal{O}(n)$, so it is the same as with Oemof. However, as already mentioned the actual time consumption is larger than for the optimization process with Oemof.

3.3.4 Oemof memory complexity

We used a similar approach to analyse the computational resource 'memory'. Due to the limitations of the computing device, the maximum timeframe consists of only 40.000 days. Otherwise, the analysed energy system stayed the same. The scalability of the memory usage was compared for ten different timeframes.

As shown in figure 3.4 the complexity class for the total simulation process is $\mathcal{O}(n)$. The memory usage grows linear with respect to the timeframe.

When reviewing the individual steps it is seen that there are differences in the scalability regarding memory usage compared to processor time. The first step, the parsing, has a

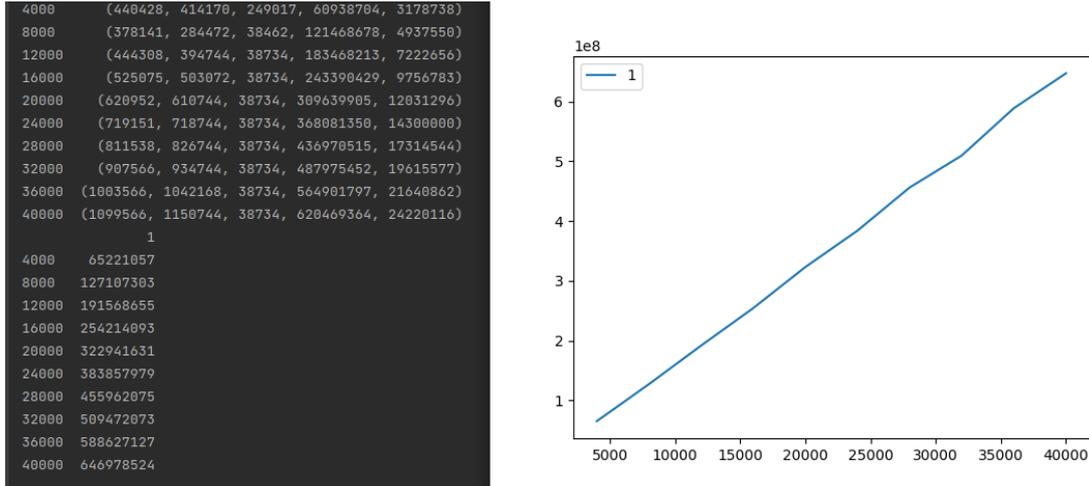


Figure 3.4: results of the Oemof memory usage measurement (own figure created with Tessif)

complexity class of $\mathcal{O}(n)$. Except for the timeframe the energy system stays the same for each simulation. Therefore it is assumed that difference follows out of the size of the list containing the timeframes. This python list grows with every timeframe and the memory usage with it.

This solidifies by looking at the results for the second step. The amount of KiB needed to run the algorithm grows by similar values compared to the parsing. The grow rate is in a range from 9.000 to 11.000 KiB when enhancing the timeframe. Therefore, we propose that the change of the complexity class is due to the growing size of the list containing the timeframe.

The transformation from the Tessif energy system to the Oemof energy system is $\mathcal{O}(1)$. It is independent from the timescale and stays the same for each simulation.

The biggest consumer of the computational resource memory space is the optimisation process, which has the complexity class $\mathcal{O}(n)$. The second biggest contributor in resource usage is the post-processing. The complexity class is again $\mathcal{O}(n)$. The usage grows from 3178738 to 24220116 KiB. Comparing the memory usage of the optimization and post-processing regarding the last timeframe, the post processing needs only about a 24th of the memory space. Compared to the optimization the influence of the other steps become mostly irrelevant.

3.3.5 PyPSA memory complexity

The analysis of the space complexity of the simulation with the modelling tool PyPSA is conducted in the same way as for Oemof. The self similar energy system was simulated ten times with different timeframes. The smallest timeframe consisted of 4.000 timesteps and the largest of 40.000. The other timeframes were evenly distributed in that range.

As we expected the first two simulation steps behave the same way as for Oemof, their complexity class is again $\mathcal{O}(n)$.

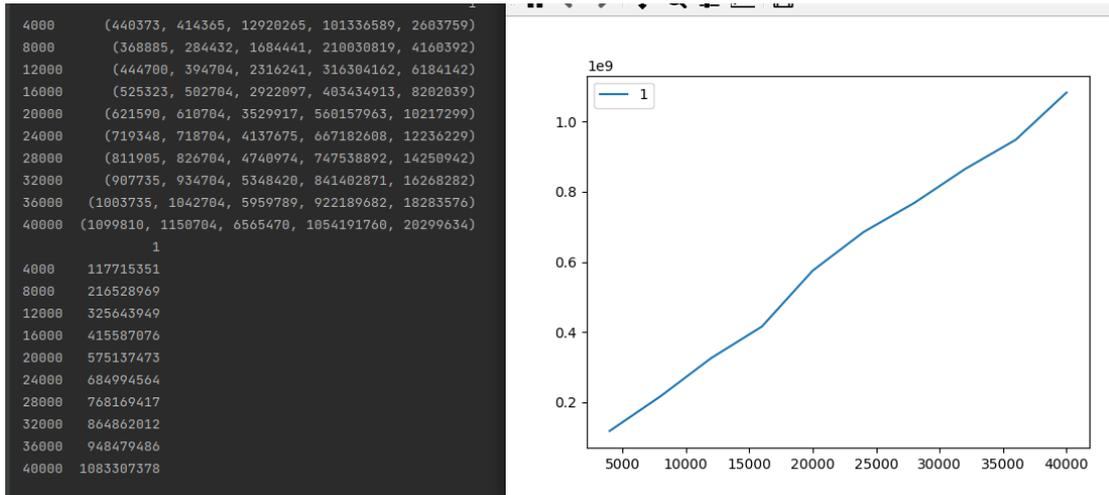


Figure 3.5: results of the PyPSA memory usage measurement (own figure created with Tessif)

Most interestingly, the transformation from the Tessif energy system to the PyPSA energy system has the complexity class $\mathcal{O}(n)$. The memory usage grows linear from 1.684.441 KiB at 8.000 timesteps to 6.565.470 KiB at 40.000.

The optimization step is by far the biggest resource consumer in this process. It needs about 50 times as much memory space as the post processing, which is the second biggest user. The complexity class for the post-processing and the optimization is again $\mathcal{O}(n)$. Summarizing, each individual step has complexity class $\mathcal{O}(n)$.

3.3.6 Comparison of Oemof and PyPSA time and space

This section compares the results of the simulation process for the modelling tools Oemof and PyPSA through the Tessif framework regarding their time and space complexity. To this end, the previously collected results are reviewed and compared.

As expected, the first two steps of the simulation have the same complexity class and very similar final results for Oemof and PyPSA. We note that the size of the list containing the timeframe has no impact on the time complexity of the execution of the first two steps, they are both $\mathcal{O}(1)$.

On the other hand, the space complexity of the first two steps is influenced by the scale of the timeframe. The complexity class of the parsing step (step 1) is $\mathcal{O}(n)$. Taking a closer look at this part of the simulation, we see that the only thing that is changing is the size of the list containing the timeframes. Therefore we assume that the change in need for memory space is directly connected to the size of the timeframes.

Looking at the transformation of the python mapping into the Tessif energy system (step 2) we see that the growth of the computational resource is similar to the growth of the preceding step, it is again $\mathcal{O}(n)$.

Reviewing the time complexity we see that the difference in the size of the list has no visible influence on the running time, but when taking a closer at the memory space complexity we see that there is a growth in the resource usage, which has to influence

the running time as well. Therefore, we predict that the actual complexity class for the time complexity for the first two steps is $\mathcal{O}(n)$ as well, but the difference is too small to be detected by the chosen time measurement library. This difference however is very small compared to the size of the optimization step that it is nearly neglectable. The memory usage of the optimization is about a thousand times higher than the usage of the parsing.

The biggest difference occurs in the third step. Whereas for Oemof this step is the fastest one and has time complexity class $\mathcal{O}(1)$ for the PyPSA simulation process this step has time complexity class $\mathcal{O}(n)$. This is a big performance difference. For the Oemof process this step takes only milliseconds independent from the timeframe. For the simulation process with PyPSA this step needs already for the second timeframe about 3.5 seconds and grows linear after that. For the last timeframe, consisting of 50.000 timesteps, the processor needs about 17.97 seconds longer.

This is solidified by the space complexity. The space complexity class for the simulation with oemof is $\mathcal{O}(1)$. The complexity class for the space complexity with PyPSA is $\mathcal{O}(n)$. However, the influence on the total memory usage is not as high as the influence on the running time performance.

As previously mentioned it is assumed that the performance difference results in the interpretation of the energy systems. Tessif and Oemof use similar approaches, whereas PyPSA differentiates here.

The optimization of both modelling tools have the same time complexity class, namely $\mathcal{O}(n)$ but there are differences in the time consumption. The optimization with PyPSA needs more time to execute and the gradient is steeper, see Figure 3.6.

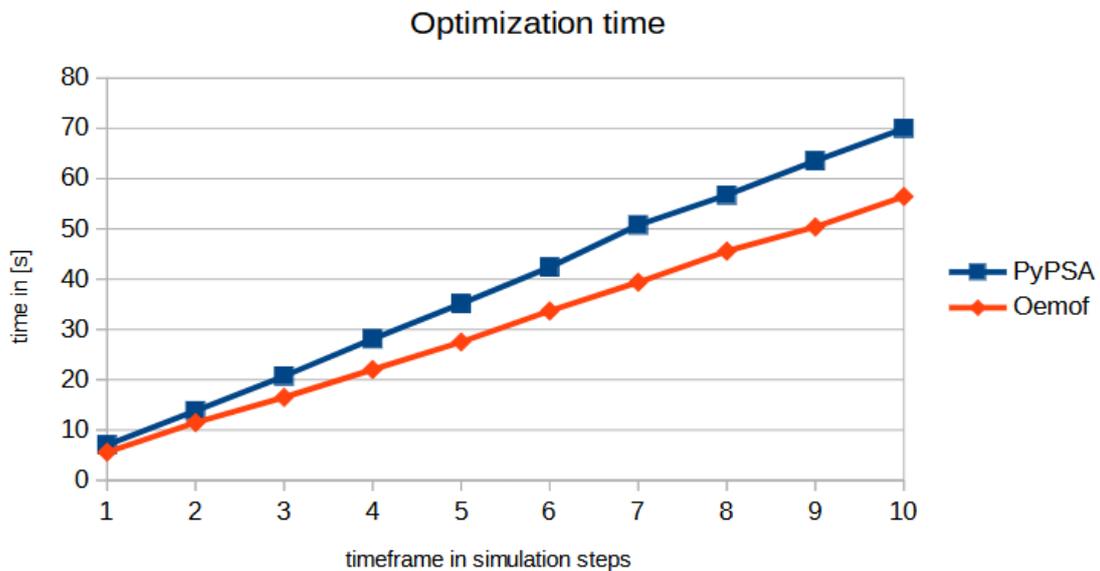


Figure 3.6: Comparison of the optimization process of the modelling tools Oemof and PyPSA (own figure)

As shown in Figure 3.6 the performance gap grows with a growing timescale.

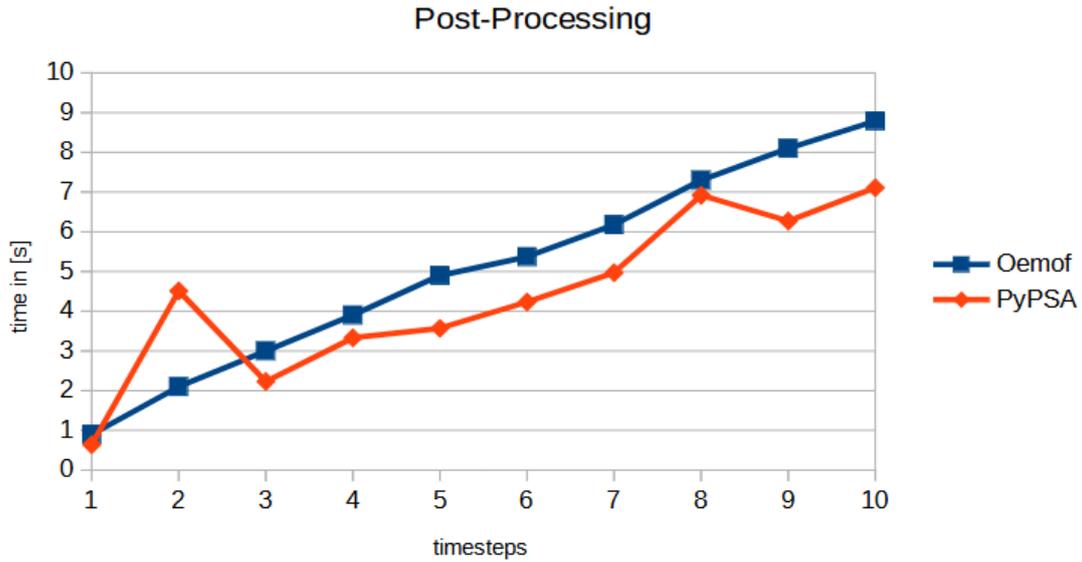


Figure 3.7: Comparison of the post-processing process of the modelling tools Oemof and PyPSA (own figure)

3.4 Scalability regarding the size of the energy system

This section analysis the scalability of the system when the size of the energy system grows. To analyse this the `asses_scalability` function was used. Throughout the analyzing process the timescale was constant. The size of the energy system was enhanced by connecting minimum self similar energy systems and creating in this way a large one.

The constant timescale that was used consists of 1.000 timesteps. The first simulation was executed with one minimum self similar energy system and with each iteration another one was connected to the energy system up until the energy system consisted of ten minimum self similar energy systems. Therefore, ten different scenarios are reviewed and compared in the following sections.

3.4.1 Expectation

In the following we analyse again the five different steps of the simulation. In order to do so we review the measurements and determine the complexity class for the time and space complexity.

As for the previously reviewed scenario with the variable timescale and fixed size of the energy system, we conjecture that the first two steps of the simulation are independent of the chosen modelling tool. However, we expect at least a complexity class of $\mathcal{O}(n)$. For every new simulation the energy system is connected to another minimum self similar energy system. Therefore the amount of data that needs to be extracted grows respectively. We conjecture a similar growth in the use of computational resources. This is expected for the transformation of the python mapping into the Tessif energy system for the same reasons.

A similar result is expected for the transformation into the tool specific energy system as well. We conjecture that the amount of resources needed to make the transformation grows with the size of the energy system. A larger energy system has more information and details stored, which have to be transformed as well. Therefore a rise in the time and memory usage is expected.

The next step is the optimization process. It is expected that this step is again the largest user of computational resources. We assume that the complexity class is at least $\mathcal{O}(n)$ and the resource consumption grows at least linear with respect to the size of the energy system.

For the post processing we expect a similar outcome as for the previous steps.

3.4.2 Oemof time complexity

Ten energy systems of different sizes were simulated. The scalability of the system regarding the size of the energy system has the complexity class of $\mathcal{O}(n)$. This means that the time grows linear with respect to the size of the energy system, see Figure 3.8.

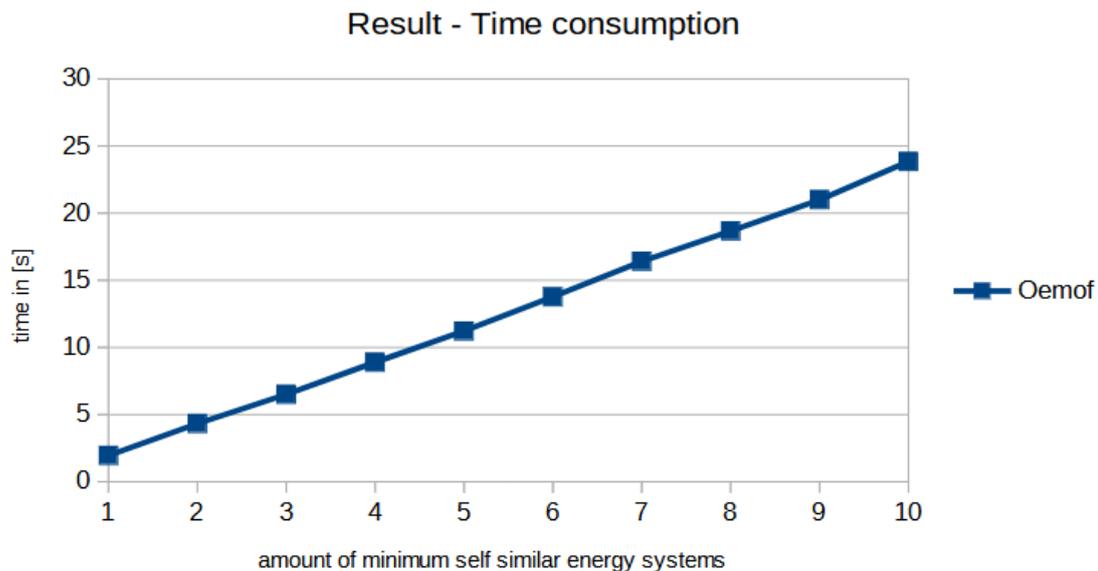


Figure 3.8: Development of the time consumption by enhancing the energy system size (own figure)

By reviewing the steps of the simulation itself we see that every simulation step has an increase in the time consumption when enhancing the energy system. The growth of the first three steps are in comparison to the post processing and especially optimization very small. However the complexity class of the first three steps is still $\mathcal{O}(n)$. The smallest step is the transformation from the Tessif energy system into the Oemof system. The execution time grows from 0.0012 seconds for the smallest system to 0.012 seconds for the largest.

The parsing process and the generation of the Tessif energy system take roughly the same amount of time and grow with the same gradient. The complexity class of $\mathcal{O}(n)$ coincides the previously made assumption about these steps.

The next step is the optimization. The complexity class is $\mathcal{O}(n)$ and it has the biggest time consumption increase when enhancing the system. We see that with a growing energy system size the influence of the first three steps become marginal in comparison to the optimization step, this is shown in Figure 3.9 as well.

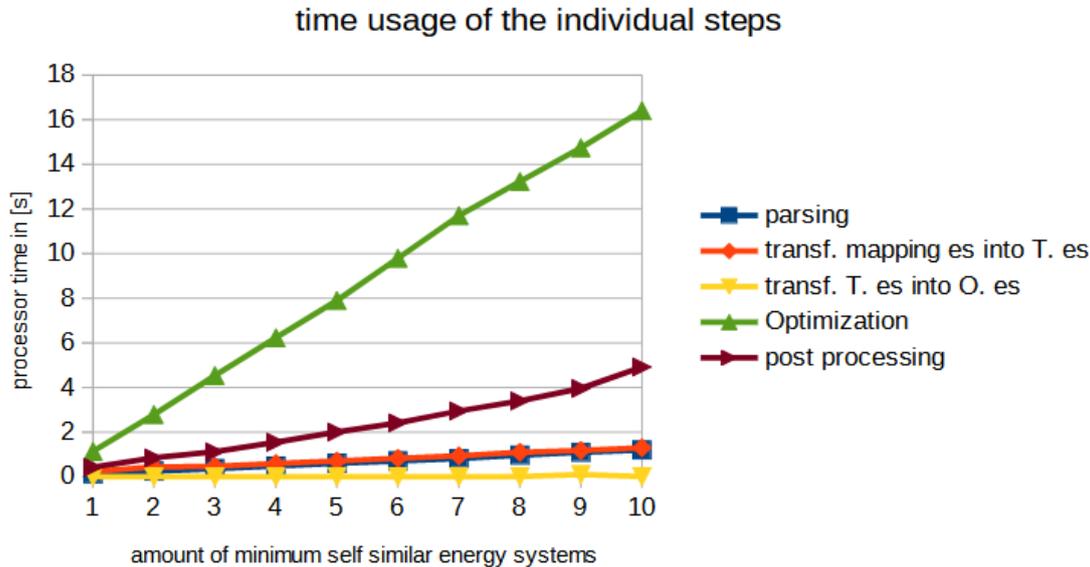


Figure 3.9: Development of the time consumption for every step when enhancing the energy system size (own figure)

The post processing is the last step. The complexity class is as well $\mathcal{O}(n)$. The gradient is steeper than for the first three steps. The post processing needs normally about 25% of the time to run the optimization.

3.4.3 PyPSA time complexity

The scenario for the analysis of the time complexity of the modelling tool was the same as the previous one for Oemof. As for the Oemof simulation the complexity class for PyPSA is $\mathcal{O}(n)$. The time grows from 2.76 seconds for one minimum self similar energy system to 20,89 seconds for the system consisting of ten connected self similar energy systems (see Figure 3.11). As we see the total time did not grow by the factor ten.

This follows from the influence of the different steps. In the beginning the gap between the time consumption of the different steps is very small. The optimization step contributes to only 53% of the used time. For the largest system in the scenario the percentage of the total time consumption grew to 68%.

The most surprising step is the transformation of the Tessif energy system into the PyPSA system. It was expected that the behaviour of the resource usage is similar or even more extreme than for the scenario with the fixed scale of the energy system size and the variable timescale. Recall that in section 3.3.3 the transformation step was consistently the second largest time consumer. As it can be seen from Figure 3.10 that this is not the case now. Therefore, the size of the timeframe seems to have a bigger influence on the running time than the size of the energy system. A possible reason could be the formats of the data sets, but more research has to be done in this area.

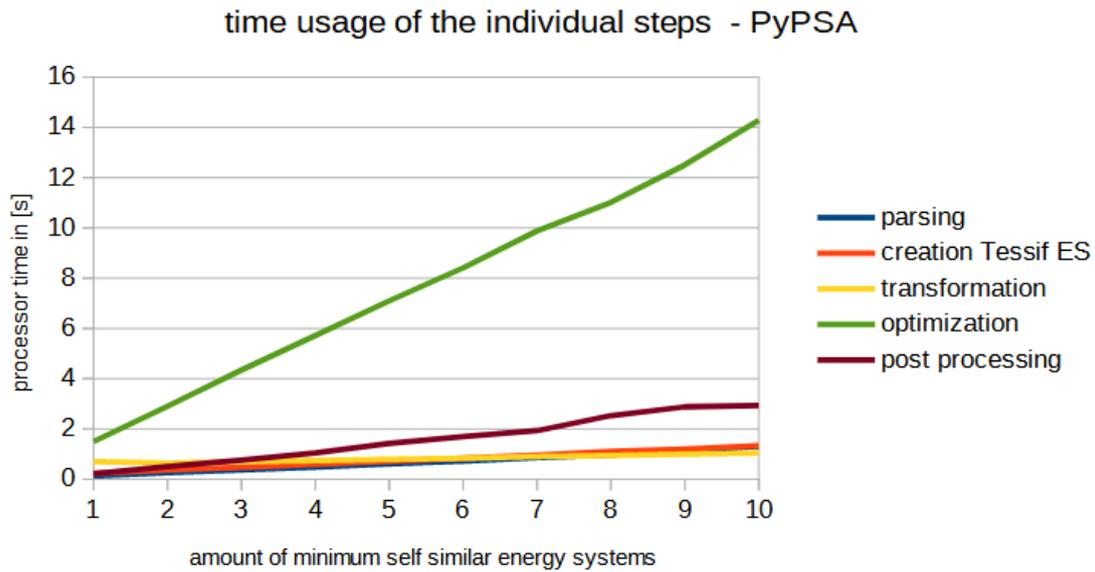


Figure 3.10: Development of the time consumption for every step when enhancing the energy system size (own figure)

Every step has the complexity class $\mathcal{O}(n)$. When enhancing the energy system, every step uses more of the computational resource time.

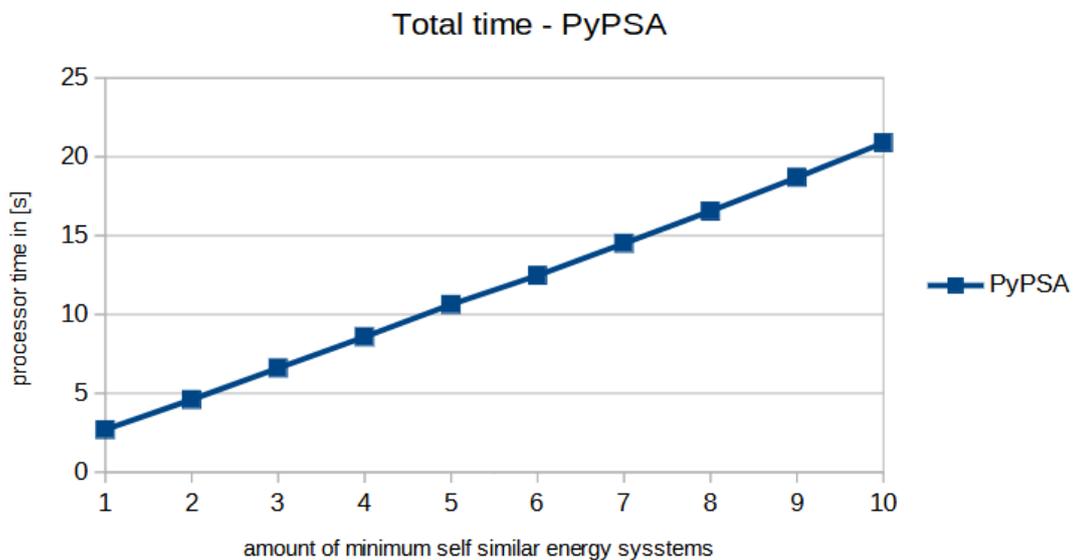


Figure 3.11: Development of the time consumption by enhancing the energy system size (own figure)

3.4.4 Oemof space complexity

The space complexity class for this scenario in Oemof is $\mathcal{O}(n)$. The memory usage to simulate one minimum self similar energy system equals approximately 21.000.000 KiB.

For every minimum self similar energy system that is connected to energy system about 21.000.000 KiB more are necessary (see Figure 3.12).

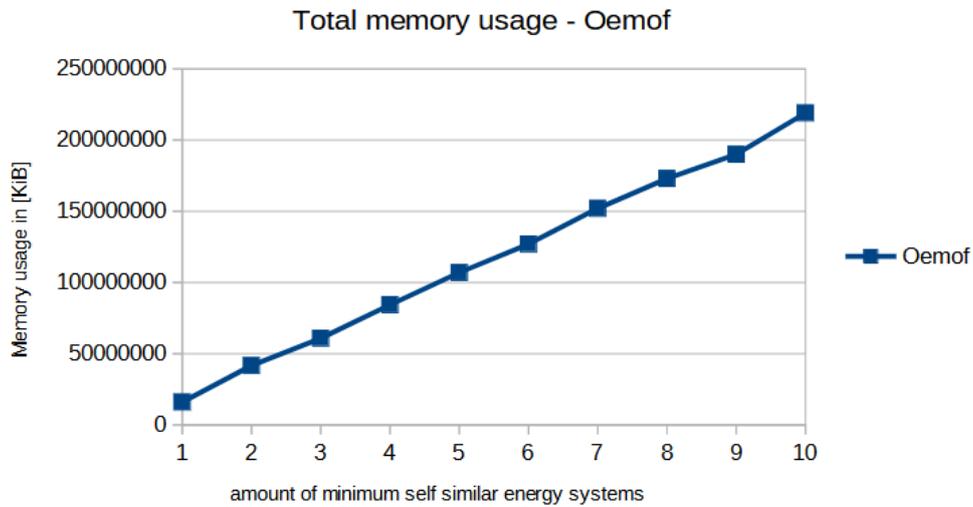


Figure 3.12: Development of the time consumption by enhancing the energy system size (own figure)

The complexity class is for every step $\mathcal{O}(n)$. Therefore the memory usage grows for every step when enhancing the energy system. The first three steps have similar memory usage.

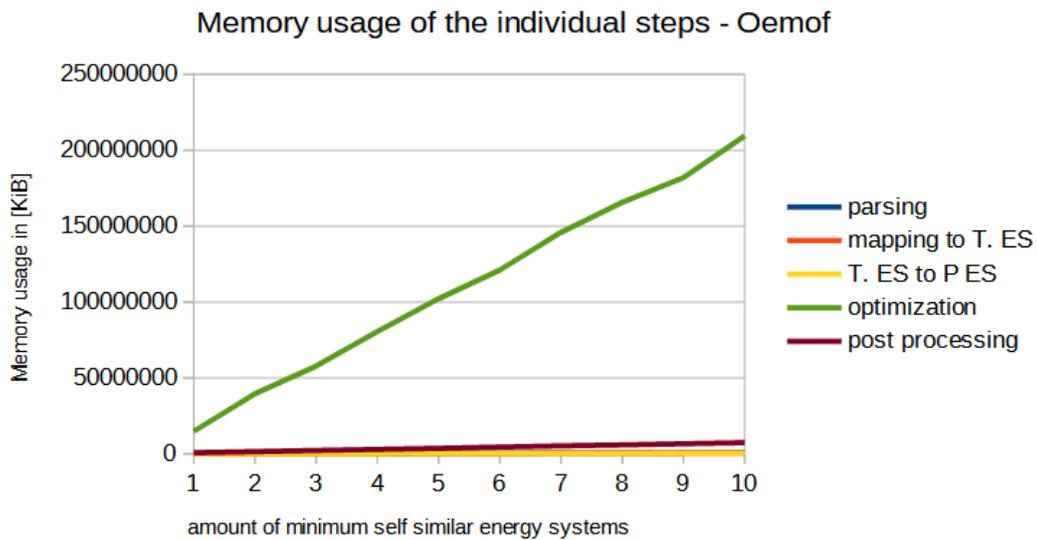


Figure 3.13: Development of the time consumption by enhancing the energy system size (own figure)

3.4.5 PyPSA space complexity

The complexity class for the memory usage in this scenario is $\mathcal{O}(n)$. The total memory usage for the first simulation with only one minimum self similar energy system needs

26909798 KiB. The largest energy system needs 264794086 KiB which is about ten times as much. It follows for the total memory usage that when we double the size of the energy system the memory usage doubles as well (see Figure 3.14).

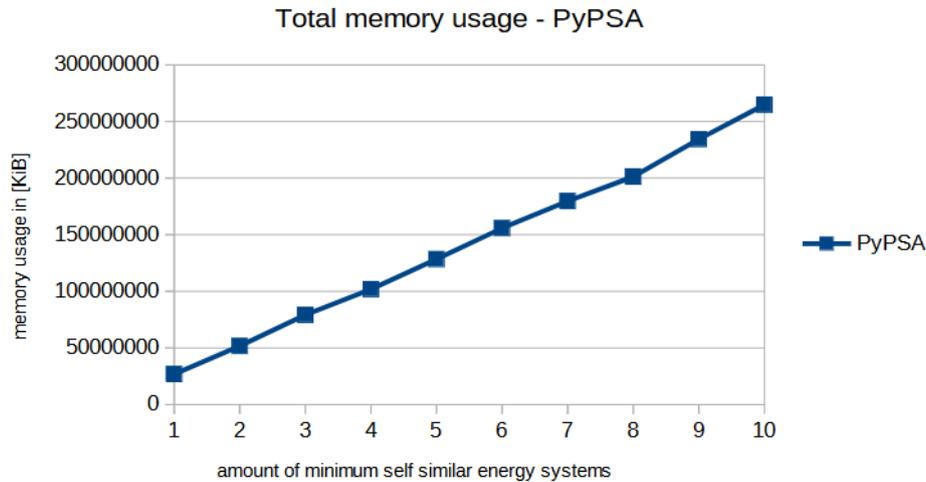


Figure 3.14: Development of the memory usage when enhancing the energy system size (own figure)

Looking at the different steps we see that almost every step has the complexity class $\mathcal{O}(n)$. The contribution to the total memory usage however is very different for each step (see Figure 3.15).

Surprisingly the memory usage of the third step, the transformation of the Tessif energy system into the PyPSA energy system, looks like it is staying constant for the different energy system sizes (for more detailed discussion see Subsection 3.4.6). This is a big difference to the simulation with the variable timeframe. As mentioned in the time complexity section for this scenario this step was the second biggest contributor there (see Section 3.3.3). This gives further evidence that the timeframe consumes more computational resources for this step for PyPSA than the scale of the energy system.

Finally, we see that the optimization step has the biggest memory usage. For the simulation with ten connected minimum self similar energy systems the optimization step contributes to 96% of the total resource usage.

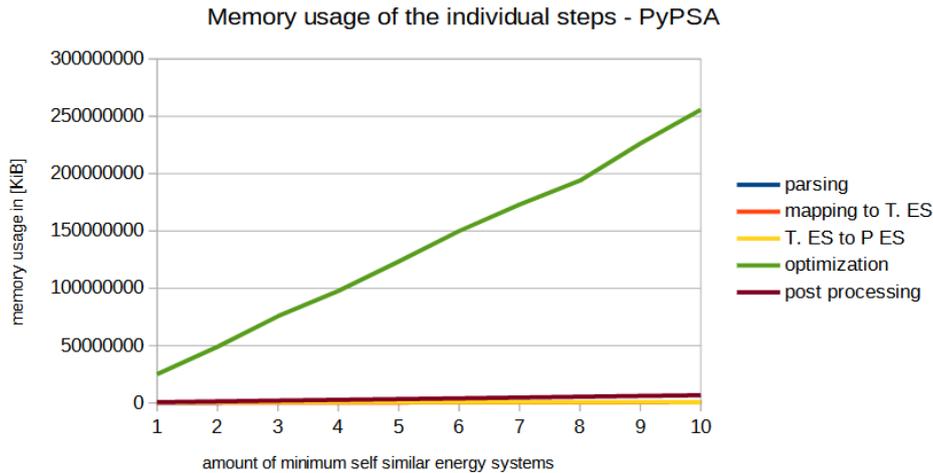


Figure 3.15: Development of the memory usage when enhancing the energy system size (own figure)

3.4.6 Comparison of Oemof and PyPSA time and space

This section analyses the previously collected data of the simulation process for the second scenario. This scenario investigates the scalability of the system with one constant timeframe and a variable size of the energy system. The `asses_scalability` function starts with one minimum self similar energy system and iterates ten times until there are ten simulations with ten different energy system sizes. Each iteration another minimum self similar energy system is connected. Therefore the last energy system consists of ten minimum self similar energy systems.

We see for the first two steps that they are independent from the chosen model. Recall that the modelling tool is up until the third step only a parameter that is passed along. The steps are however not independent from the size of the energy size. If the size of the energy system is enhanced there is also more data stored in the external data which has to be extracted. This results in a longer running time and a higher memory usage. Same goes for the transformation of the mapping into a Tessif energy system. This is coherent for the time and memory usage analysis.

When looking at the third step of the simulation from the Oemof modelling tool we see the expected results. The results grow linear with respect to the size of the energy system. This is true for the memory usage as well as for the running time. When doubling the size of the energy system the amount of computational resources doubles as well. The results for modelling tool PyPSA however deviate from the one we expected, especially after seeing the results from the first scenario with the variable timeframe and the fixed size of the energy system. In the last scenario the third step was the second largest consumer of computational resources. However when enhancing the size of the energy system the usage of computational resources is nearly constant. The time grows from the smallest to the largest energy system by the factor 1.67. The transformation takes for one minimum self similar energy system 0.68 seconds and for ten connected 1.04 seconds. For Oemof on the other hand the time needed for the first transformation is only 0.0012 seconds. The running time however grows for every new connected minimum

self similar energy system by 0.0012 seconds. Therefore it grows after ten iterations by the factor ten.

Reviewing the memory usage this can be seen even clearer. Looking at figure 3.16 we see that there is no visible difference in the plot for PyPSA. The Oemof modelling uses in the beginning a lot less from the computational resource space, but it grows accordingly to the size of the energy system. We assume that it is easier for the computing device to find the data it needs to transform the energy system for the modelling tool Oemof. However the actual transformation is faster for the modelling tool PyPSA. The modelling tool PyPSA uses mainly dataframes from the pandas library which are very fast to process. Oemof however uses different data formats as mappings and dictionaries which can result in a difference of the running time and memory usage. This results in a faster transformation for the modelling tool Oemof for smaller energy systems vice versa a faster transformation for larger energy systems for the modelling tool PyPSA.

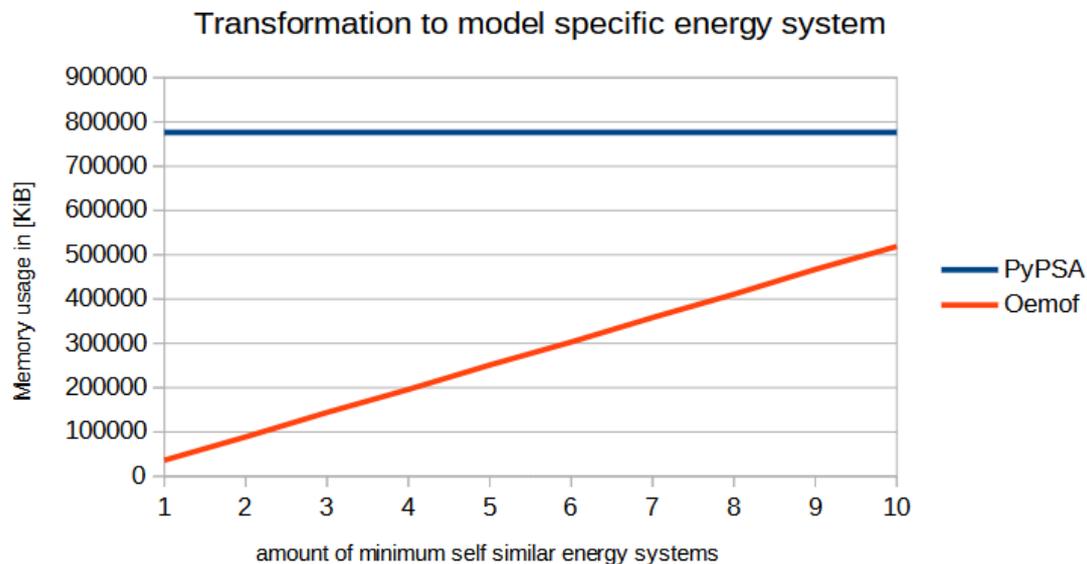


Figure 3.16: Comparison of the memory usage of the third step (own figure)

The optimization step (see Figure 3.17) is by far the biggest consumer of the computational resources. As previously mentioned it is responsible for 96% of the memory usage for the modelling tool PyPSA for the largest energy system. For the modelling tool Oemof this step is responsible for 95.7%. The PyPSA plot has a steeper gradient regarding the memory usage.

However, when we look at the running time in Figure 3.18 we see that the optimization is faster for the PyPSA modelling tool despite the higher memory usage. The incline for the Oemof tool is steeper than for PyPSA. Oemof is faster for the two smallest energy systems but afterwards PyPSA is faster.

The last step we analyse is the post processing. The time complexity class for both modelling tools is $\mathcal{O}(n)$, but Oemof needs about twice as much time for every energy system. The difference in the memory usage is not that extreme but it is still lower for PyPSA. We assume that the performance advantage results from the used data formats used by the modelling tools. As already mentioned for step number three PyPSA mainly

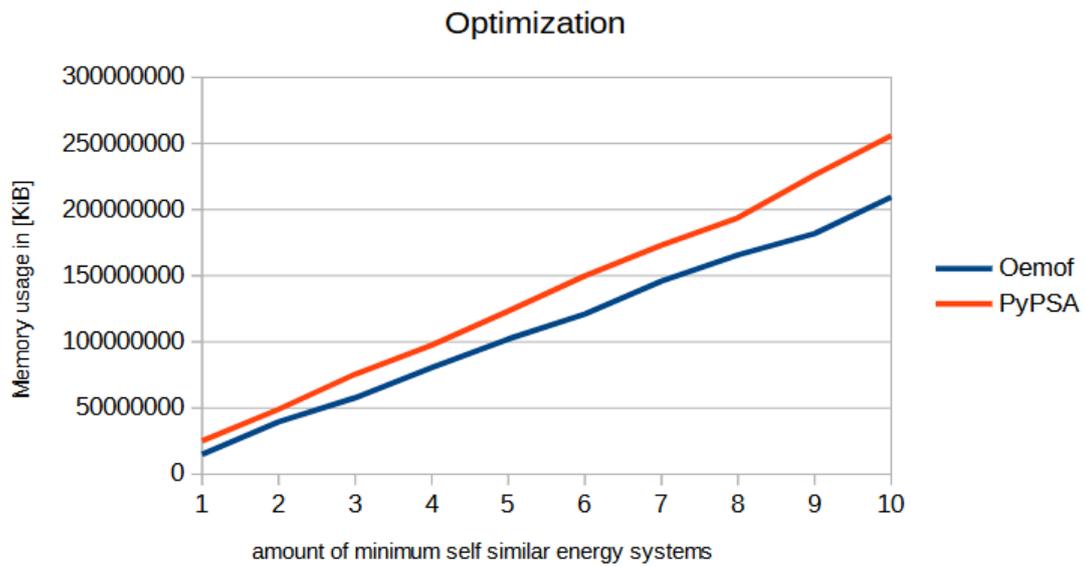


Figure 3.17: Comparison of the memory usage of the third step (own figure)

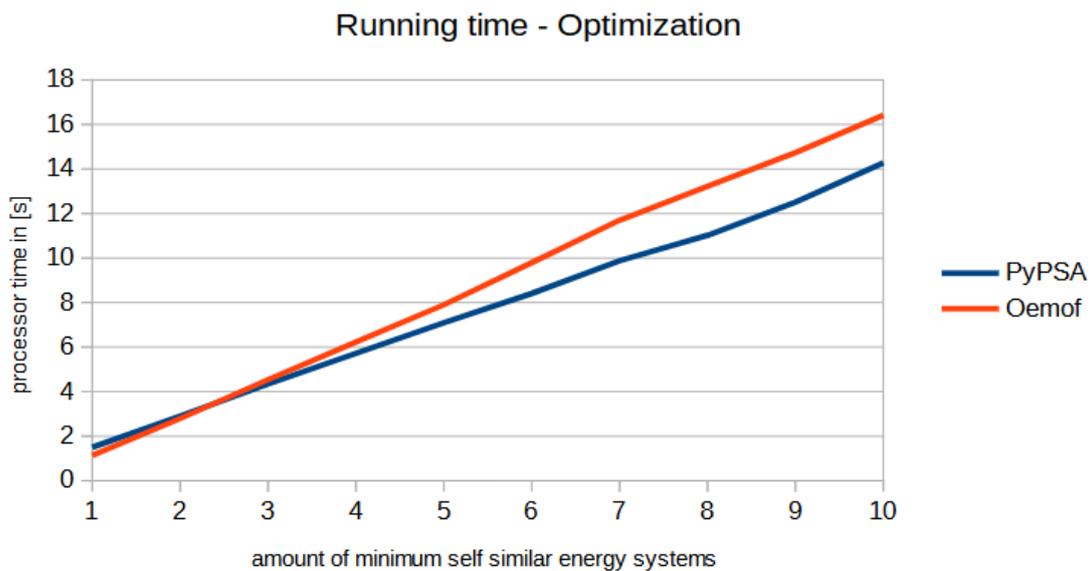


Figure 3.18: Comparison of the memory usage of the third step (own figure)

uses the dataframes from Pandas, which are very fast to process. This could result in a faster running time.

In conclusion after comparing both modelling tools regarding the size of the energy system PyPSA has an advantage over Oemof when it comes to the computational cost except for the memory usage in the optimization step. The trend predicts that the gap grows even wider if the energy system is even more enhanced.

3.5 comparison timescale and scale of the energy system

The scenario for the last comparison contains a growing timescale as well as an enhancing energy system. To analyze this the `asses_scalability` function is used. The largest energy system will consist of three connected minimum self similar energy systems. Therefore the `asses_scalability` function will iterate three times and build three different energy systems. Each of these systems will be simulated with four different timeframes. The first timeframe consists of 2.000 timesteps, the second of 4.000, the third of 6.000 and the last of 8.000. Therefore, there are in total 16 different simulations for each modelling tool in the end.

We expect that the higher order complexity class of every step from the previous two scenarios is dominating when enhancing both variables. For example, if for one step the time complexity class is $\mathcal{O}(n)$ for the scenario with the variable timeframe and $\mathcal{O}(1)$ for the scenario with the variable energy system, then we expect the complexity class to be $\mathcal{O}(n)$ when enhancing both. And when both steps have the same complexity class we expect that the class stays the same but that the rise in computational resources grows steeper.

Furthermore, the contribution of every simulation step to the total resource usage is analyzed.

3.5.1 Results

From our measurements we see that our conjecture about the complexity class and the rise in the resource usage was right. First, we take a more detailed look at the results for the modelling tool Oemof.

We see in Figure 3.19 that the use of computational resources rises when enhancing the scale of the timeframe or the size of the energy system. In addition, the figure shows that enhancing the size of the system mainly influences the optimization part, whereas increasing the timeframe also has a visible influence on post processing. Furthermore, the bigger the energy system is, the larger is the influence of the timeframe. In other words, the effect of increasing the size of the timeframe on the resource usage is bigger for a large energy system than for a small one. This could be explained by the fact that the gradients of the two previously introduced scenarios are intertwined. This is as expected, because when enhancing both variables more connected data is generated.

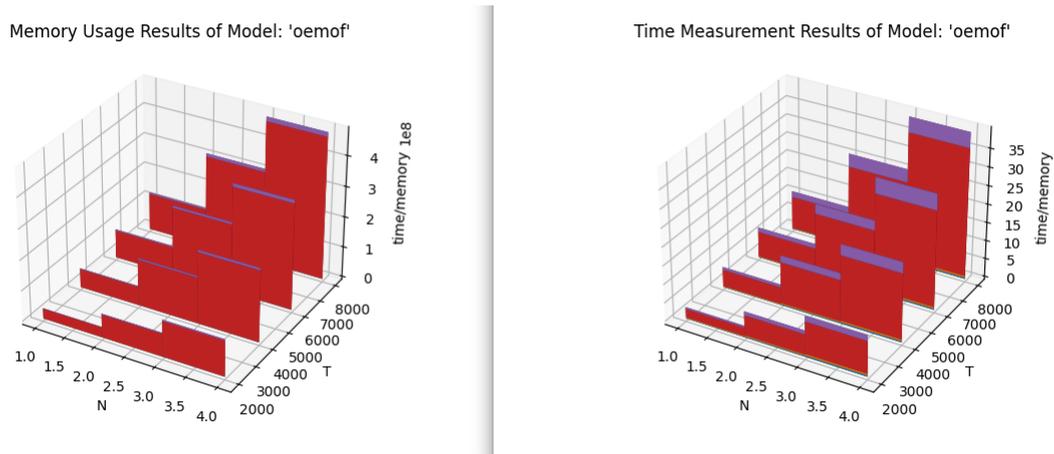


Figure 3.19: stacked 3D plot - Oemof (own figure created with Tessif)

We see from the stacked 3D plot for the memory usage (Figure 3.20) that the optimization process is by far the biggest contributor and has the highest gradient. Finally, the complexity class is $\mathcal{O}(n)$ when enhancing both variables at the same time.

We now take a look at PyPSA. In figure 3.20 we observe more visible differences than in the Oemof case. The biggest one, as happened before in the scenario with the variable timeframe, is the third step. Possible reasons are stated in 3.4.3. As before, this step is mainly influenced by the timeframe and not so much by the energy system size.

Comparing both plots for the memory usage and the time measurement we see that the steps executed in the framework Tessif, namely the extraction of the external data, the transformation into the Tessif energy system, the transformation to the PyPSA energy system and the post processing do have an influence on the running time. However, the influence of the integrated modelling tool is a lot larger. Furthermore, the gap grows when enhancing the size of the energy system and the scale of the timeframe. We expect that this trend continues for bigger systems. Looking at the memory usage this is even more extreme. The other steps are barely visible or even invisible next to the optimization.

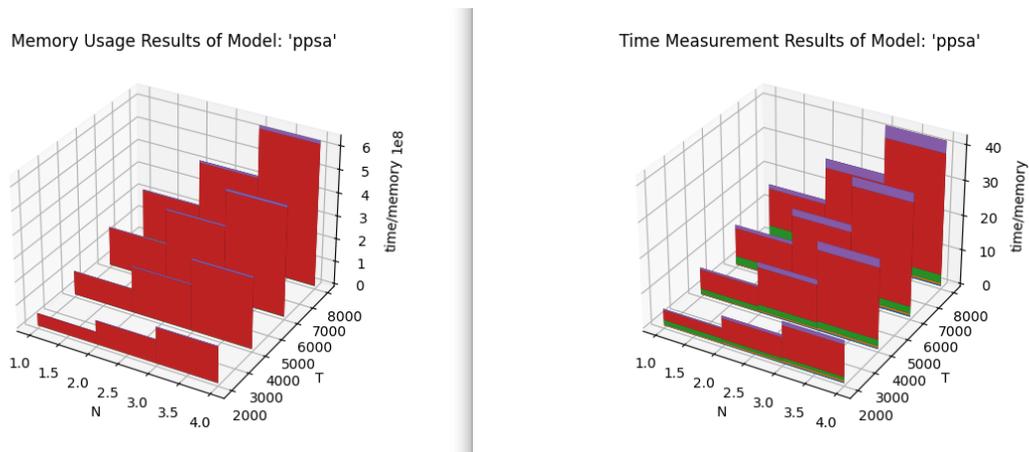


Figure 3.20: stacked 3D plot - PyPSA (own figure created with Tessif)

Finally, we take a closer look at the simulation with three connected energy systems

and a timeframe with 6.000 timesteps. We noticed in the stacked bar plot that the optimization step is by far the largest consumer when it comes to memory usage.

Comparing figure 3.21 and 3.22 we see that the biggest difference is, as previously mentioned, step three. Whereas for Oemof this step is the fastest one, PyPSA needs significantly more time. Otherwise the structure is for both tools similar. For the optimization and the post processing there is a slight performance advantage for the modelling tool PyPSA. However, the main point is that the optimization step is for the running time as well the largest contributor. For the Oemof simulation the optimization takes roughly 40 seconds whereas all the other steps combined only need about 5 seconds. This means that the optimization step is responsible for about 90% of the processor time. It is different for the modelling tool PyPSA. The optimization takes about 35 seconds and all the other steps combined about 10 seconds. This results in a percentage of 72%. This is mainly due to the faster running time in combination with the increased time needed to transform the Tessif energy system into the PyPSA energy system. As we have seen in Chapter 3, the percentage of the optimization step also increases when increasing the variables. Consequently, by enhancing the system the impact of the steps executed directly by the modelling framework Tessif become more and more neglectable.

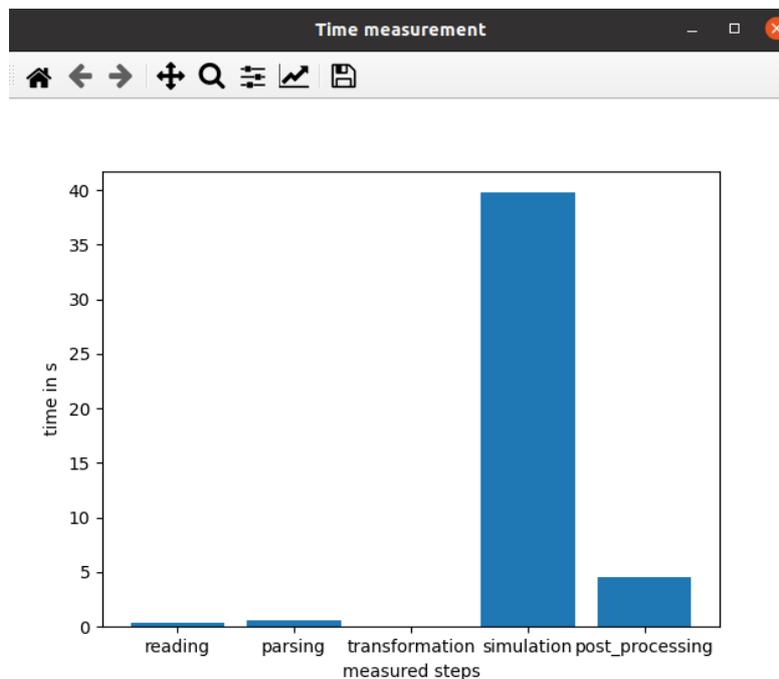


Figure 3.21: time measurement bar plot - Oemof (own figure created with Tessif)

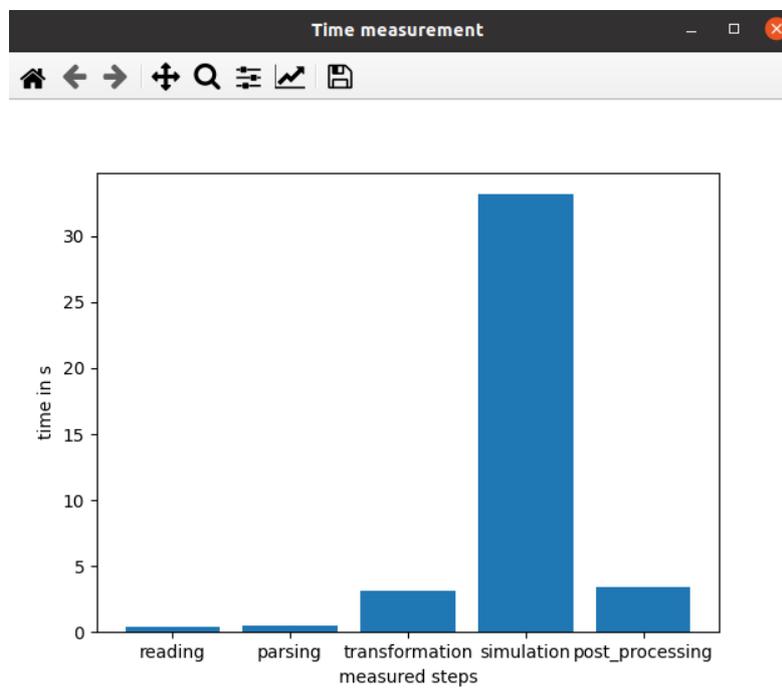


Figure 3.22: time measurement bar plot - PyPSA (own figure created with Tessif)

4 Conclusion and Outlook

This thesis is concluded by a short summary followed by a conclusion. Furthermore there will be a short outlook on other modeling tools and the possibility of parallelization.

4.1 Conclusion

In the process of this thesis several methods were implemented in Tessif to analyse the time and space complexity of energy system simulation tools. To this end, an energy system was designed, where the size of the system and the scale of the timeframe can be adjusted by the user. This energy system does not resemble any real system, it is designed for the sole purpose of analysing the scalability of modelling tools.

The main results are the following: The complexity class of the entire simulation process is $\mathcal{O}(n)$ for both computational resources (time and memory) and both modelling tools (PyPSA and Oemof). The optimization process is the largest consumer of those resources and has also the fastest incline when enhancing the size of the system or the scale of the timeframe. In comparison to the optimization process, the other simulation steps become almost irrelevant when increasing the input size. Thus, the main percentage of the resources are used by the modelling tools integrated in Tessif and not by Tessif itself.

Finally, we want to mention that the third step (the transformation into the model specific system) behaves quite differently depending on the modelling tool. For example, when we treated the timescale as a variable this step had complexity class $\mathcal{O}(1)$ in Oemof and $\mathcal{O}(n)$ in PyPSA.

To make the analyses of this thesis even more accurate, simulating very big energy systems could be of interest. This, however, was not possible because of the performance limitation of the computing device that was used.

4.2 Outlook

To address the problem of the performance limitation, a parallelization approach can be used. Parallelization is enhancing the amount of processing elements, which could reduce the running time significantly. This would be important for the optimization step of modelling very large energy systems. By analysing these, the required changes to adress the climate change can be identified.

In addition to improving the performance through more sophisticated computing devices or the parallelization, new energy system modelling tools can be implemented and analysed in Tessif.

Bibliography

- [1] J. HOUGHTON, G. JENKINS, and J. EPHRAUMS: *Climate change: the IPCC scientific assessment*. Cambridge University Press, for Intergovernmental Panel on Climate Change.
- [2] *Klimafolgen und Anpassung*. (accessed on 6 March 2022). URL: <https://www.umweltbundesamt.de/themen/klima-energie/klimafolgen-anpassung>.
- [3] INTERGOVERNMENTAL PANEL ON CLIMATE CHANGE: *Climate Change 2014: Mitigation of Climate Change: Working Group III Contribution to the IPCC Fifth Assessment Report*. Cambridge University Press, 2015.
- [4] *Deutsche Klimaschutzpolitik*. URL: <https://www.bmwk.de/Redaktion/DE/Textsammlungen/Industrie/klimaschutz.html>.
- [5] S. RATH-NAGEL and A. VOSS: “Energy models for planning and policy assessment”. In: *European Journal of Operational Research* 8.2 (1981), pp. 99–114.
- [6] H.-K. RINGKJØB, P. HAUGAN, and I. SOLBREKKE: “A review of modelling tools for energy and electricity systems with large shares of variable renewables”. In: *Renewable and Sustainable Energy Reviews* 96 (Aug. 2018), pp. 440–459.
- [7] AMMON: *Tessif: Transforming Energy Supply System (Modelling) Frameworks. Dokumentation*. unpublished. Institut für Energietechnik, Technische Universität Hamburg.
- [8] *A Modular Open Source Framework to Model Energy Supply Systems*. (accessed on 6 March 2022). URL: <https://oemof.org/>.
- [9] *Python for Power System Analysis*. (accessed on 6 March 2022). URL: <https://pypsa.org>.
- [10] S. ARORA and B. BARAK: *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [11] G. VAN ROSSUM and F. L. DRAKE JR: *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.
- [12] *time — Time access and conversion*. (accessed on 12 April 2022). URL: <https://docs.python.org/3/library/time.html>.
- [13] *timeit — Measure execution time of small code snippets*. (accessed on 12 April 2022). URL: <https://docs.python.org/3/library/timeit.html>.
- [14] *tracemalloc — Trace memory allocations*. (accessed on 12 April 2022). URL: <https://docs.python.org/3/library/tracemalloc.html>.
- [15] *psutil documentation*. (accessed on 12 April 2022). URL: <https://psutil.readthedocs.io/en/latest/>.
- [16] O. FORSTER: *Analysis 1*. Springer Fachmedien Wiesbaden, 2016.
- [17] H.-J. MITTAG: *Statistik*. Springer Berlin Heidelberg, 2014.