Showcase of Energy Modelling with Julia

Navin Kumar Trivedi, Benjamin Vilmann, and Aditya Vyas

January 2021, TUM

Abstract

Julia is a relatively new, cost-free, and open-sourced programming language from 2012 and is specifically promising for numerical computing. This report will show the capabilities within Julia and suggest a best-practice toolchain in the domain of optimization problems for energy modelling purposes.

The optimization problem will consist of the *Unit Commitment*. The model mimics the overall power system in Germany and it's objective is thus to dispatch power plants at each state of various generator type. In this way, the model can be evaluated next to real data from the German Bundesnetzagentur and congestion issues in the transmission system can be addressed.

Since this is a showcase of energy modelling with Julia, the energy model compromises, however, its virtue to simulate reality in order to focus on capabilities in Julia and various optimization options in general.

Name	**Mat. No.**	**Abbr.**
Navin Kumar Trivedi	03722319	N
Benjamin Vilmann	03739843	B
Aditya Vyas	03722506	A

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

AML	Algebraic Modelling Language
GHG	Green House Gases
IDE	Integrated Development Environment
IPCC	Intergovernmental Panel on Climate Change
LP	Linear Programming
MILP	Mixed-Integer Linear Programming
MINLP	Mixed-Integer Non-Linear Programming
NLP	Non-Linear Programming
QP	Quadratic Programming
RES	Renewable Energy Sources
SDP	Semi-Definite Programming
SOCP	Second Order Conic Programming
UC	Unit Commitment

Symbols

η_E	Efficiency of transformation of mechanical energy into electrical energy
η_H	Efficiency of transformation of chemical / nuclear energy into heat
η_T	Efficiency of transformation of heat into mechanical energy
η_{other}	other thermal, mechanical and electrical losses, and self consumption
η_{PP}	Power plant efficiency
a	Technology dependent plant correction factor
b	Technology dependent plant correction factor
$C1$	Ambient conditions dependent plant correction factor
$C2$	Plant age correction factor
c_i	Unit fuel costs including CO_2 emission costs
$D1$	Technology dependent optimum efficiency correction factor
$D2$	Technology dependent optimum efficiency correction factor
$P(i,t)$	Generation of the unit i in time t
P_D^t	Power Demand at time t
$P_i^{downmax}$	Maximum power below which the unit cannot immediately change without taking time called Running Down ramp rate
P_R^t	Spinning reserve (Reserve capacity needed during emergency operating conditions or load swings) at time t

P_i^{upmax} Maximum power above which the unit cannot immediately change without taking time called Running Up ramp rate

P_{max} Maximum generating capacity

P_{min} Minimum generating capacity

P_{nom} Nominal loading of the plant

P_{opt} Loading at optimum efficiency of the plant

$t_i^{downmin}$ Minimum down-time

t_i^{down} Down-time of unit i

t_i^{upmin} Minimum Up-time

t_i^{up} Up-time of unit i

x_i^t Unit activation variable

1 Introduction

The main focus of this project lab is to show the possibilities of energy modelling within the programming language, Julia. Therefore, the report will deal with a optimization problem that can vary from simple to complex, namely the *Unit Commitment Problem*. This will endure the report to outline the optimization features of Julia. Since it is a showcase, the report allows itself not to get caught up with detailed theoretical aspects of the energy model itself. Furthermore, the report does not emphasise on absolute cost values of the energy model, rather discusses the energy mix as the result of the model.

In section 3, aspects of Julia relevant for the solution of optimization problems will be covered including documenting results hereof. Section 4 presents the *Unit Commitment Problem*. Section 5 documents the implementation and is followed by section 6, which comments on the results. Lastly, section 7 concludes on the showcase and the possibilities within Julia for energy modelling.

The implemented code is available on Github [1].

2 Problem statement

The objective of this project lab is to develop an energy model in Julia and to demonstrate its capabilities of energy modelling. This energy model mimics the German power system that consist of conventional power plants and an increasing amount of renewable energy systems (RES). The model is based on the *unit commitment* problem, which provides many possibilities to implement constraints of various complexities.

The criteria for a satisfying model is a solution with a comparable energy mix with respect to the actual generation data. The data used for modelling is obtained from Bundesnetzagentur, SMARD.DE [2] and "Kraftwerklist" [3]. This data is used to derive the parameters such as capacity limits and demand, which aids in developing an optimized solution. This optimized solution is then compared with the real generation data for both the summer and the winter scenario of 2019.

3 Julia

Julia is a new open-source programming language from 2012, which tends to bring the best of two worlds together: speed and intuition. Typically, programming languages are either considered fast, low-level languages (like C, C++, and Fortran), or slow, high-level languages (like Python, MATLAB) [4]. Julia is therefore a very potent language for numerical computing in many different scientific fields as it supports various computation methods, e.g. *differential equations, machine learning*, and *optimization (mathematical)*. Julia is on a rise in popularity and moved up 24 indices to #23 from #47 on the Tiobe index on a single year [5, 6].

This section will further elaborate relevant Julia syntax and present tools in Julia and IDE:

3.1 Choice of IDE

Three IDEs have been tested for understanding the best applicability in Julia, namely Jupyter Notebook, Microsoft Visual Studio, and Juno (an Atom package). In the table below, central parameters for user experience and editing support has been checked for each platform:

Table 1: IDE parameters

	VS-Studio	Atom (Juno)	Jupyter
Code completion	✓	✓	✓
Cost-free	✓	✓	✓
Debugger	✓	✓	✓
Document mode			✓
Github integration	✓	✓	✓[1]
Online editor			✓
Open-source	✓	✓	✓
Plot pane	✓	✓	✓[2]
Syntax highlighting	✓	✓	✓
Tabular data viewer	✓		

There are of course additional features for each platform that can be crucial for the personal choice of an IDE like available themes for the user interface but this will not be covered further in this report.

3.2 Syntax

As Julia is a hybrid between the two paradigms of programming languages, most of the Julia syntax is comparable to languages like C/C++, Python, R, and MATLAB. The documentation outline the noteworthy differences to the respective languages [7]. In addition to that, some noteworthy similarities and significant syntaxes also deserve to be adressed:

Splat & Slurp, ... (N)

The ... operator is one of the versatile features in Julia and has a dual functionality: It can be used either as a splat operator or as a slurp operator.

When used as a splatter, ... operator splits one argument into several different arguments in a function call. It is very convenient as it enables the user to write functions taking an arbitrary number of arguments. Such type of functions that use splatting operators are called **varargs** functions, i.e. "variable number of arguments" [8]. The splat operator can be placed after the last positional argument in the function call. Generally, when using the splat operator, a tuple of values is spliced into the varargs call precisely where the variable number of arguments go. Moreover, the iterable object splatted into a function call need not be a tuple always. Furthermore, the function that the arguments are splatted into do not need to be a varargs function (although it often is).

[1] There is an extension that integrates Github: `https://github.com/jupyterlab/jupyterlab-github`
[2] Plots are part of the document in Jupyter and not specifically in a plot pane.

However when ... operator is used in the function definition instead of function call, it acts as a slurp operator. The slurp operator ... can be used to combine many arguments into one arguments in function definitions. The use of ..., for combining many different arguments into a single argument, is called **slurping** [9].

The following piece of code explains the slurping function of the ... operator in a simple way:

```
function varargs_slurp(args...)     #Function Definition (Slurping)
        return args
end

varargs_slurp(1,2,3)  #Function call

Output => (1,2,3)
```

The following piece of code explains the splatting function of the ... operator in a simple way:

```
function varargs_splat(a::Int64, b::Int64, c::Int64, x...) #Function Definition
        w = a+b+c
        println("$x is the argument")
        println(" $w is the sum ")
end
x = [4,5,1,2,1,7]
varargs_splat(x...)   #Function call (Splatting)

Output => (2, 1, 7) is the argument
          10 is the sum
```

Pipe, `|>` (N)

Piping (or "chaining") in Julia allows to combine multiple function calls making them more precise and compact for reading. It avoids saving intermediate results without having to embed function calls within one another. With the pipe operator `|>`, the code to the right hand side of `|>` operates on the result from the code to the left hand side of it. In other words, what is on the left becomes the argument of the function calls that is on the right. Thus, piping is very useful in data manipulation and improves readability for functional oriented languages like Julia [10].

Pipes are very limited in the `Base`, as it only support functions with one argument and only a single function at a time. However, the `Pipe` package along with the `@pipe` macro allows the `|>` operator to use functions with multiple arguments and multiple functions.

The following piece of code explains the function of piping operator in a simple way:

```
add(a) = a+8;
mul(a) = a*5
div(a) = 4/a;
a=2 |> add |> mul |> div |> println   #Piping add, mul and div

Output => 0.08 # a = 2 => div(mul(add(2))) = 4/(2+8)*5) = 0.08
```

Symbol, `:` (A)

The essence of symbol in Julia is associated with its support for metaprogramming and can be understood from the way Julia processes a code [11]. A program code begins as a string, which is parsed into an object called expression. An expression consists of two parts - symbol and argument. Here the symbol is an expression identifier or an interned string identifier. The character `:` serves two purposes. Firstly it serves to create symbol or interned string used in expressions and secondly it serves to create expressions without using the constructor `Expr` explicitly. This is termed *quoting*. The following section of code depicts the same:

```
julia> :x                           # Creating symbol
:x
julia> typeof(ans)
Symbol

julia> x = :(a+b)                   # Quoting
```

```
    :(a + b)
    julia> typeof(ans)
    Expr
```

Also the syntax `Symbol()` can be used to concatenate strings into symbol as depicted by the following code section

```

Symbol("Julia","_","Energy","_","Modelling")
:Julia_Energy_Modelling
```

However, the true essence of a symbol in Julia is to represent a variable in metaprogramming [12].

Matrix notation (A)

Unlike most technical languages, Julia does not treat arrays in a special way. The array library is almost implemented in Julia itself and is just treated like any other code written in Julia. The compiler allows programs with the use of arrays to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times. However, the syntax and workspace in Julia is comparable to that in MATLAB. In addition to extensive support for multi-dimensional arrays, Julia provides a native support to some of the common linear algebra operations such as transpose, determinants, inverse, eigenvalues and factorization for a variety of matrices/ special matrices, which can be loaded with `using LinearAlgebra`. The following code shows some of the matrix notations:

```
using LinearAlgebra
A = [1 2 3; 4 5 6; 7 9 6]    # Matrix-A          3x3
O = ones(3,3)                # Unit matrix       3x3
Z = rand(3,2)                # Random matrix     3x2
E = I(2)                     # Identity matrix   2x2
F = det(A)                   # Determinant of A  1x1
H = inv(A)                   # Inverse of A      3x3
```

"Bang" convention, `!` (B)

It is a convention in Julia to append `!` to a function to distinguish if a function will change content of the input or not. This is because functions normally are *passed-by-sharing* [13, 14]. Passed-by-sharing means that the input value of the function is not copied but a new binding is created to the value and refers to identical values to the inputs. Example:

```
    a = [1,2,3]              # creates array: [1,2,3]
    push!(a,4)               # Pushes 4 to array, so a = [1,2,3,4]
```

Here, the "bang" notation is needed to tell Julia that the original array is changed and no new bindings are created. To clarify the intuition, the bang notation is not optional for methods: There is for an example not an equivalent function called `push()`, which then would make a copy of the array, `a`, with the appended value, `4`.

Macros, `@` (B)

Much coding can be trivial and repetitive tasks are often handled by functions in many cases. Julia also provides *metaprogramming* by macros [11], which basically copy-pastes code. The practical use of metaprogramming are disputed because it only benefits special cases and same coding desire can be achieved by recursion [15]. Details of macros will not be further elaborated, but the notation will be remarked because macros are widely used in mathematical computation packages in Julia (JuMP). Following is a example:

```
    @show 2 + 2
    # prints "2 + 2 = 4" and the REPL evaluation

    @show ones(2,2)
    # prints "ones(2, 2) = [1.0 1.0; 1.0 1.0]" and REPL evaluation
```

The macro, `@show`, shows in other words the quoted expression used in the input and equals that to the evaluation sent to the compiler.

3.3 Packages

Julia has evolved as an unprecedented combination of several existing languages like R, Python and MATLAB, thus becoming a multipurpose language. As in January 2021, a total of 4809 packages are available in Julia [6], making it a very efficient platform for performing optimization problems, statistical analysis, machine learning and many more. To handle all this, Julia provides an easy way to import packages and build them in scripts or the REPL. There are two ways to fetch a package:

```
# 1st approach:
Pkg.add("PackageName")
# 2nd approach: Enter Pkg REPL by typing ']' and then write:
add PackageName
```

The package handler is also well documented [16].

3.3.1 Mathematical Optimization

The possibilities for performing optimization problems in Julia are numerous as a large set of packages are available, that can support vast range of optimization problems. The primary optimization package used in Julia is the Julia integrated `JuMP.jl`. In addition to `JuMp.jl`, Julia also offers `JuliaSmoothOptimizers`, which is an Infrastructure for Continuous Optimization in Julia, and `JuliaNLSolvers`, which offers support for standard optimization algorithm associated with unconstrained or box-constrained problems [17, 18].

JuMP.jl

JuMP is an open-source AML, which is embedded in Julia. Formerly, JuMP was a part of JuliaOpt and now has outgrown and replaced JuliaOpt. JuMP is now a feature in Julia that widens the application and reach of Julia and makes Julia an appealing option for scientific computation. While being an alternative to commercial AMLs, JuMP takes advantage of several features in Julia and delivers a very capable ecosystem for modelling or scientific computation in general. JuMP extends beyond AML and serves as a tool for scientific computing in the direction of scientific domain specific languages and efficient derivative computation [19].

Being an AML, JuMP functions as an interface between the high level algebraic syntax and optimization routines called solvers. It currently supports a number of open-source and commercial solvers for a variety of optimization problem classes. JuMP communicates with most solvers in memory, avoiding the need to write intermediary files. It uses a generic solver-independent interface provided by the MathOptInterface package, making it easy to change between a number of open-source and commercial solvers. JuMP also includes `Convex.jl`, an algebraic modeling language for convex optimization based on the concept of Disciplined Convex Programming [20].

Having a capable embedded AML, a large selection of optimization specific solver packages are available. Table 2, shows few solvers, their licensing and functionalities. The solvers used in this project laboratory are Cbc, CPLEX, Ipopt and Juniper.

Table 2: Solvers and their functionalities

Solver	Package	License	LP	QP	SOCP	MILP	NLP	MINLP	SDP
Artelys Knitro	KNITRO.jl	Commercial	✓		✓	✓	✓	✓	
BARON	BARON.jl	Commercial					✓	✓	
CDCS	CDCS.jl	Open Source	✓		✓				✓
Cbc	Cbc.jl	Open Source				✓			
CPLEX	CPLEX.jl	Commercial	✓		✓	✓			
Gurobi	Gurobi.jl	Commercial	✓		✓	✓			
Ipopt	Ipopt.jl	Open Source	✓	✓			✓		
Juniper	Juniper.jl	Open Source	✓			✓	✓	✓	
MOSEK	Mosek.jl	Commercial	✓		✓	✓			✓
NLopt	NLopt.jl	Open Source	✓				✓		
SCIP	SCIP.jl	Open Source	✓			✓	✓	✓	

3.3.2 Data handling

Julia provides a range of packages for data handling and manipulation. In addition to some of the very popular and well-supported data science packages like DataFrames.jl for working with tabular data, and ExcelFiles.jl for reading and editing data from excel files, Julia also provides some other great libraries for specific purposes including MySQL.jl to access MySQL from Julia, and JuliaDB.jl for analysing database in Julia. Among the vast available libraries for data science in Julia, this model particularly aims in utilizing three packages namely DataFrames.jl, ExcelFiles.jl, and FileIO.jl. The functionalities of these packages are explained briefly in the following sections.

DataFrames.jl

DataFrames.jl provides a set of tools for working with tabular data in Julia. The design and functionality of DataFrames.jl resembles to those of `pandas` (in Python) and `data.frame`, `data.table`, and `dplyr` (in R), making it a great general purpose data science tool, especially for those who have a prior knowledge of R or Python.

DataFrames.jl plays a crucial role in the Julia Data ecosystem, and has strong integration with a range of different libraries [21]. It is well integrated with different data science libraries essentially for data-wrangling, data manipulation, and import-export of different files. The below is the list of few libraries used in this energy model:

- **DataFramesMeta.jl:** It includes a range of convenience functions like `select` and `transform` to provide a user experience similar to that provided by `dplyr` in R.
- **Query.jl:** It has advanced data manipulation capabilities for DataFrames. Query.jl provide commands that can filter, project, join, flatten and group data from a DataFrame.
- **Import/Export files:** DataFrames.jl work well with a range of formats, including CSVs (using CSV.jl), Stata, SPSS, and SAS files (using StatFiles.jl), and reading and writing parquet files (using Parquet.jl).

The code for DataFrames.jl is very simple and straightforward. An illustration for reading and selecting the specific columns of .xlsx file is as shown below and the figure 1 represents the outcome of this code.

```
using ExcelFiles, DataFrames, DataFramesMeta
df = load((raw"c:\\Users\\navin\\Desktop\\generators2.xlsx"), "Data2") |> DataFrame
df2 = @select(df, :Pnom, :min)
```

ExcelFiles.jl

The ExcelFiles.jl package provides functionality to read data from an Excel file into a DataFrame. This package uses the Python xlrd library [22]. It acts an adapter library for ExcelReaders.jl.

The following packages are also available for reading the excel files:

- **XLSX.jl:**Uses purely Julia libraries
- **Taro.jl:**Uses java libraries
- **ExcelReaders.jl:**Uses python xlrd libraries

An interesting feature of ExcelFiles.jl which differentiates it from the other packages is that it is well integrated with the Queryverse.jl (a meta package that pulls together a number of packages for handling data in julia). Furthermore, it has a **load** function which can handle iterable tables and load the excel files into dataframes, alongside supporting the pipe syntax which is very useful when combining the excel files with the queries [23].

	Pnom	min
	Float64	Float64
1	10.0	4.5
2	10.0	4.5
3	648.0	291.6
4	944.0	424.8
5	628.0	282.6
6	297.0	133.65
7	295.0	132.75
8	15.0	6.75
9	875.0	393.75
10	875.0	393.75
11	640.0	288.0
12	465.0	209.25
13	857.0	385.65
14	465.0	209.25
15	9.3	4.185

Figure 1: DataFrames

FileIO.jl

The FileIO Package provides a common framework for detecting file formats and dispatching to appropriate readers/writers. This package belongs to the **General IO** subcategory under the **FileIO** main category of the Julia Packages. `FileIO.jl` consists of two important functions: `load` and `save`. In comparison to Julia's inbuilt low-level read and write support, this package provides high-level support for formatted files through `load` and `save` functions [24].

3.3.3 Plotting

Programming languages with down-the-shelf packages for plotting options refer typically to either only one package or have one dominant plotting package. In Julia, there seems to be starting competition between two: Namely Plots.jl [25] and Makie.jl [26].

To show the differences between the two major plotting packages, a grid plot (2x2) each with a *pie chart*, *bar chart*, *line plot*, and a *scatter plot* has been discussed.

Plots.jl

In figure 2a, a decent plot is generated from a fairly simple code:

```
1 using Plots                                          # Import package
2 ax1 = pie(1:3,title="Pie")                           # Pie chart
3 ax3 = plot(rand(3,3),title="Line")                   # Line plot
4 ax2 = bar(1:3,title="Bar")                           # Bar chart
5 ax4 = scatter(rand(3,3),title="Scatter")             # Scatter plot
6 p = plot(ax1,ax2,ax3,ax4,                            # Grid plot
7     layout = grid(2, 2, heights = [0.5,0.5], widths = [0.5,0.5]))
```

Plots.jl is quite intuitive: In the first line, the package is imported, and then several plots are assigned to each plot-object. These plots can be shown as subplots in an another plot, which is done in line 6.

line 2 to 5, it is seen that the plot type is compatible with many input types, e.g. it accepts `AbstractRanges`, like `1:3`. It also accepts matrices, e.g. the `rand(3,3)` creates a 3×3 matrix of random values, $\mathbb{R} \in [0;1]$. The philosophy of data interpretation in Plots.jl is that the columns represents the plotted series [27], which is an important note for labelling data as an example.

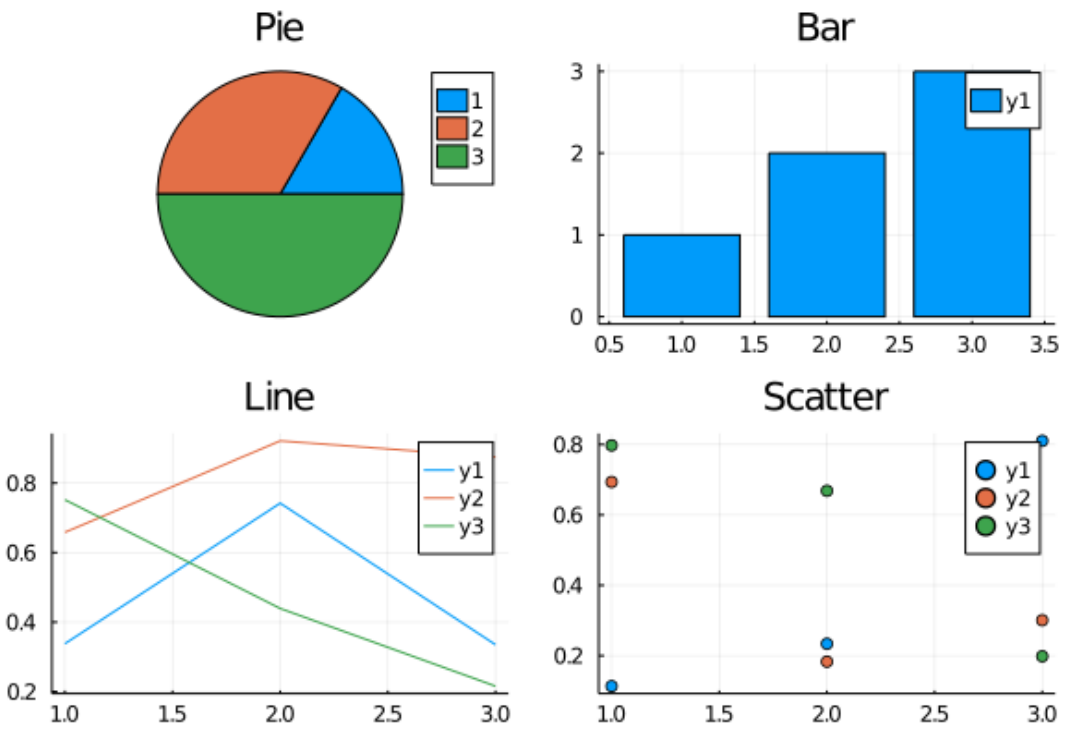
`Plots.jl` also handles the general formatting of the plot, i.e. colors are being distributed and a legend is added. So, with few readable lines of code, a lot is actually done.

However, `Plots.jl` have some drawbacks regarding customization, e.g. custom legends are not supported at this moment. There is a gallery showing the capabilities with `Plots.jl` for further interest [28].

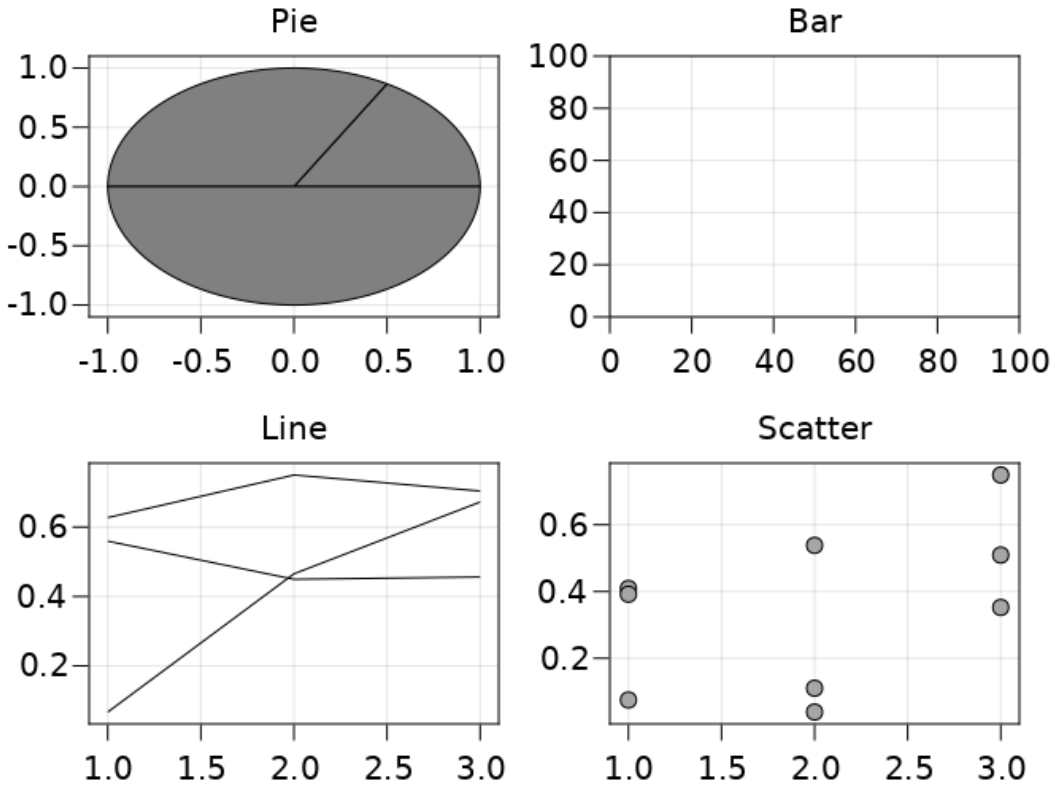
Makie.jl

In figure 2b, a less appealing plot comes to vision invoked by a slightly more complex coding:

```
using CairoMakie                                       # Import packages
using AbstractPlotting
scene, layout = layoutscene()                          # Preparing grid plot
ax1 = layout[1, 1] = LAxis(scene,title="Pie")          # Pie chart
pie!(ax1,[i for i in 1:3])
ax2 = layout[2, 1] = LAxis(scene,title="Line")         # Line plot
lines!(ax2,rand(3))
lines!(ax2,rand(3))
lines!(ax2,rand(3))
ax3 = layout[1, 2] = LAxis(scene,title="Bar")          # Bar chart
# Bar plot is not possible
ax4 = layout[2, 2] = LAxis(scene,title="Scatter")      # Scatter plot
scatter!(ax4,rand(3))
scatter!(ax4,rand(3))
scatter!(ax4,rand(3))
```



(a) Plots.jl



(b) Makie.jl

Figure 2: Comparison of Julia plot packages

In the first line of the code, the backend for the plotting package is imported[3]. The next line imports the package and then the plot is generated by various lines of code. It should be noted that the grid is generated by the `layoutscene()`, which then is populated. This then requires that the plot is itself an argument in the plotting function. This makes it a little more difficult to read. However, the `layoutscene()` feature is quite intuitive because less definitions of grid dimensions are required from the user in the general case[4].

Makie's plotting functions do not accept matrices nor ranges. Therefore, multiple lines are required to pass the data series to the respective `LAxis()`. In addition, Makie.jl does not map colors for the data series associated for each axis nor generates a legend by default.

Furthermore, the pie chart is ellipsoid and keeps the axis details and is thus far away from delivering a publishable plot. Also, a function for a bar chart is not implemented at this moment.

Beside the lacking features, colors and legends can easily be added with additional code. The legend can also be customized and `Makie.jl` is a very competent plotting package for more complex plotting options in general, e.g. for 3D-plots. It also supports animation and delivers interactive tool buttons. All of these features can be found documented in the *Makie Gallery* [31].

As a sidenote,`Makie.jl` is sponsored by the German *Federal Ministry of Education and Research* [26].

Comparison

`Plots.jl` is the most engaged plotting tool in Julia[5], which might be due to the plain and simple functionality, and it is simple to use for general plotting cases. It is intuitive and swiftly implemented. But if it comes to customization, then `Makie.jl` could be the plotting package to go with – if the more complicated data structure is found read- and writeable, and the proper functions are implemented. All the plots in this report are done with `Plots.jl` excluding the plots in figure 2b, 5, and flow charts in figure 9.

3.4 Community

Julia is open-source and therefore, an active community is necessary to progress and develop. Few central platforms for discussion and knowledge sharing for development processes and technical issues are noted for reference:

Table 3: Julia communities and platforms for support

Discourse	`https://discourse.julialang.org/`
Github	`https://github.com/JuliaLang/`
Stackoverflow	`https://stackoverflow.com/`
Documentation	`https://docs.julialang.org/en/v1/`

This report is also based on several engagements on these communities. [32, 33, 34, 35, 36, 37, 38, 39, 40, 41].

[3] It seems to be a general theme for plotting packages in Julia to have a dynamic choice of backend. However, Plots.jl says it is to make a bigger variety of features and plotting styles available for the user [29]. Backends will not be covered further in this report.

[4] There is some good documentation and a tutorial on this [30].

[5] This assumption is based on the count of "forks" on GitHub as an expression of the engagement from the Julia community [25, 26]

4 Energy Model

This section focuses on the theoretical aspects of Unit Commitment problem and provides an overview of the various parameters and constraints required to solve them in Julia.

4.1 Unit Commitment

Unit Commitment deals with the idea of planning and dispatching power generation to match the energy demand at the optimized cost. This is done for a specified time interval while satisfying all the given constraints. The model can include storage facilities or just satisfy a load balance criteria.

The committed unit should satisfy the forecasted system load and reserve requirements at minimal operating cost, subject to a large set of other system, technical, and environmental constraints. Depending on the shares of various power plants in the generation system, fairly different cost functions and site conditions arise in mathematical models for unit commitment. This energy model considers the generation from various conventional energy sources along with the current scenario of huge participation of renewable energy sources in Germany.

Typically, the UC problem is solved by using Mixed Integer Linear Programming if the objective function and the constraints can be linearized. However, there are several other methods like Lagrange Relaxation and Heuristic methods proposed for solving UC problems. As both linear and non-linear cost functions are considered in this energy model, therefore both the MILP and the MINLP methods have been tested for generation optimization under the UC problem. Moreover, the implementation of the transport model has been done using the mixed integer linear programming method.

Objective Function

The objective function considered in this UC problem is the minimization of generating cost including the cost of CO_2 emissions. The generating cost occur mainly due to the fuel consumption and can be represented as a linear function of production level.

A simple objective function for linear programming is expressed as follows:

$$\min \sum_{t=1}^{T} \sum_{i=1}^{I} c_i \cdot P_{G_i} \tag{1}$$

where,

- I = No. of generators
- T = No. of time-steps
- P_{G_i} = Generation of i_{th} generator
- c_i = Fuel cost of i^{th} generator (€/MWH) + Emission cost of CO_2 (Penalty cost(€/tonne) × Emissions (tonne/MWH))

The following objective function includes a variable for mixed integer programming:

$$\min \sum_{t=1}^{T} \sum_{i=1}^{I} \left(c_i \cdot P_{G_i} \cdot x_i^t + S(P_{G_i}, x_i) \right) \tag{2}$$

where,

- x_i = unit activation variable (binary) for time step t for i^{th} generator
- c_i = Fuel cost of i^{th} generator (€/MWH) + Emission cost of CO_2 (Penalty cost(€/tonne) × Emissions (tonne/MWH))
- Start Up Cost, $S(P_{G_i})$

Table 4: Constraints

Constraints	Constraints Sub-division	Expression
Unit Constraints	Maximum generation capacity	$P(i,t) < P_{max}$
	Minimum stable generation	$P(i,t) > P_{min}$
	Minimum Up-time	**If** $U(i,t) = 1$ **and** ${t_i}^{up} < {t_i}^{upmin}$ **then** $U(i,t+1) = 1$ **where** $U(i,t) = 1$ means Unit is **ON**
	Minimum down-time	**If** $U(i,t) = 0$ **and** ${t_i}^{down} < {t_i}^{downmin}$ **then** $U(i,t+1) = 0$;**where** $U(i,t) = 0$ means Unit is **OFF**
	Running Up ramp rates	$P(i,t+1)$ $P(i,t) \leq P_i^{upmax}$
	Running Down ramp rates	$P(i,t)$ $P(i,t+1) \leq P_i^{downmax}$
System Constraints	Load-Generation Balance	$\sum_{i=1}^{N} P_{G_i}.{w_i}^t = {P_D}^t$
	Spinning Reserve	$\sum_{i=1}^{N} P_{G_i}.{w_i}^t = {P_D}^t + {P_R}^t$
Environmental Constraints	CO_2 Emissions	
Network Constraints	Transmission Capacity Transmission Loss	
Cost Constraints	Start-up cost	$S(P_{G_i} \cdot x_i)$
	Running Cost	

4.1.1 Constraints

Constraints are the limitations in power systems in order to avoid any serious problems. These constraints can be related to the unit limitations or technical limitations or can also be environmental limitations.

The constraints in the UC problem can be classified as below:

1. Unit Constraints
2. System Constraints
3. Environmental Constraints
4. Network Constraints
5. Cost Constraints

Table 4 on page 11 provides a detailed summary of the typical constraints considered in the UC problems.

However, for a more simplistic approach, this energy model considers only some of the constraints mentioned in the table 4. These include maximum generation capacity, minimum stable generation, load-generation balance, CO_2 emissions cost, transmission capacity and transmission losses of the line, and the running cost and start-up cost.

4.1.2 Non-linear partial load

The modelling of UC problem can be made further accurate by including conventional power plant efficiency. Following partial load plant efficiencies are usually quadratic and bi-quadratic functions of loading and are referenced to empirical validation. The curves are characterized by a few typical points associated with optimal loading, efficiency at optimal loading, minimum and maximum loading and their efficiencies [42]. This consideration also provides an opportunity for better integration of emission constraints. This changes the terms of the objective function for the conventional power plants, based on parameters such as plant technology, the plant age, and ambient conditions. These considerations result in a non-linear behavior of the cost function.

This cost function primarily considers plant efficiency. The efficiency of a conventional power plant is a product of efficiencies of different subsystems of the plant. These subsystems have their own unique efficiency characteristics [42].

$$\eta_{PP} = \eta_H \cdot \eta_T \cdot \eta_E \cdot \eta_{other} \tag{3}$$

The relation between plant efficiency and loading is developed considering the characteristic point based on empirical observations from the plant curve such as optimal loading, efficiency at optimal loading, nominal loading as well as plant correction factors $a, b, C1, C2$ and η_{opt}. Equation 4, depicts efficiency as a function of loading [42]:

$$\eta(P_G) = \eta_{opt} \cdot C1 \cdot C2 \cdot \left[1 + a\left(P_{ratio} - \frac{P_G}{P_{nom}}\right) + b\left(P_{ratio} - \frac{P_G}{P_{nom}}\right)^2\right] \tag{4}$$

Where,

- $P_{ratio} = \frac{P_{opt}}{P_{nom}}$
- $C1 = -0.0025 t_a + 1.0375$, where t_a is annual average ambient temperature at the plant location
- $C2 = S_2\tau^2 + S_1\tau + S_0$, where τ is plant age
- $\eta_{opt} = D_1 ln(P_{nom}) + D_2$

The factors $a, b, P_{ratio}, D_1, D_2, S_0, S_1$ and S_2 can be referred from the Table A.1 in the appendix section.

The heat rate of a plant is simply the inverse of plant efficiency and describes the amount of energy expended in developing useful work, Equation 5, shows heat rate as a function of loading:

$$HR(P_G) = \frac{1}{\eta_{opt} \cdot C1 \cdot C2 \cdot \left[1 + a\left(P_{ratio} - \frac{P_G}{P_{nom}}\right) + b\left(P_{ratio} - \frac{P_G}{P_{nom}}\right)^2\right]} \tag{5}$$

The non-linear partial load cost function is simply a product of plant loading, heat rate function and unit fuel cost as depicted by equation 6:

$$F(P_G) = P_G \cdot HR \cdot c = \frac{k_1}{\left(\frac{k_2}{P_G}\right) + (k_3 \cdot P_G) + k_4} \tag{6}$$

Where,

- $k_1 = \frac{c}{\eta \cdot C1 \cdot C2}$
- $k_2 = 1 + a \cdot P_{ratio} + b \cdot (P_{ratio})^2$
- $k_3 = \frac{b}{(P_{nom})^2}$
- $k_4 = \frac{a - (2 \cdot b \cdot P_{ratio})}{P_{nom}}$

Figure 3, shows the non-linear partial load cost function for two thermal power plants with different technologies - Gas turbines and Steam turbines and their linear equivalent as a function of loading up to 100 MW.

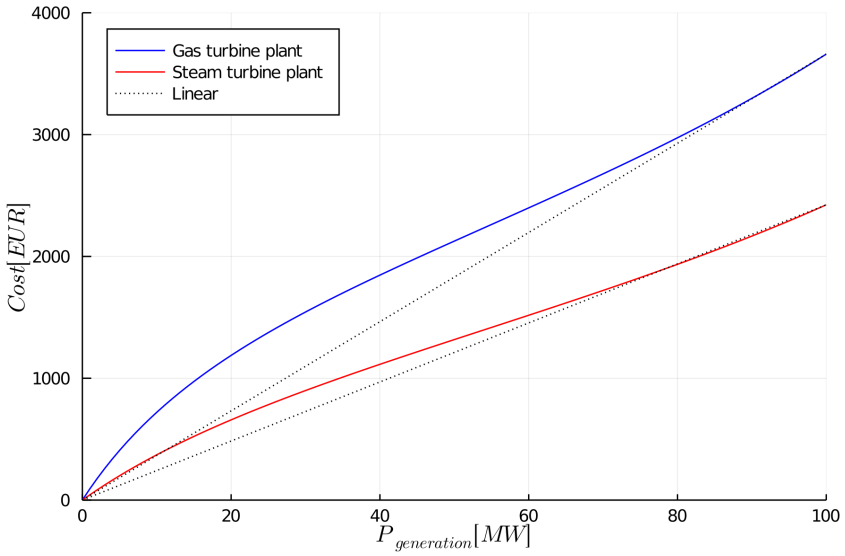


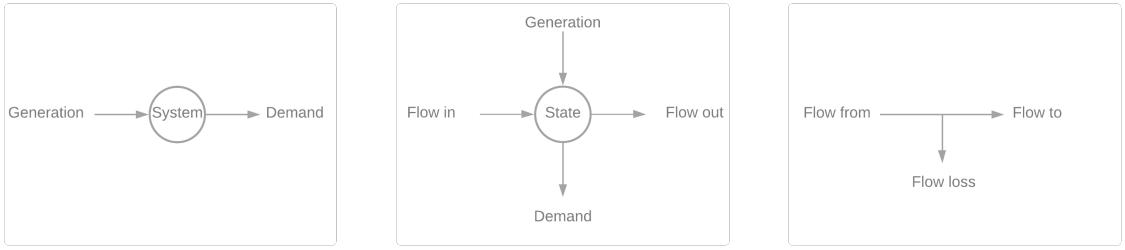
Figure 3: Non-linear partial load cost function

4.1.3 Transport model

In energy systems, transport models are highly relevant due to the inherent need for transmission of power from production site to demand site. In AC systems, the Optimal Power Flow (OPF) can be calculated in various ways and there has over time been development of many different algorithms to formulate the power flow. However, optimizing the power flow is computation intensive and requires parameters of voltage at the busbars and line impedances. This seems to be overwhelming regarding the purpose of the report – but it is still crucial to implement transmission constraints in the system!

Therefore, the system is represented by a graph, $G(V,E)$, with vertices (from now on nodes) and edges (from now on lines) [43]. The nodes represent the site of production and demand, and the lines represent the transmission with a capacity constraint and a loss parameter.

The main objective of the simple UC problem is to generate power at the lowest cost with the constraint to generate power equal to the demand. When including a transport model into the UC problem, the power now must be generated and distributed at the lowest cost and must of course still satisfy the load balance. However, two new constraints are required to be satisfied, namely the *node balance* and the *flow balance*:



(a) Load balance **(b)** Node balance **(c)** Flow balance

Figure 4: Transport model constraints

The following constraints can be formulated from the nodal analysis of figure 5:
Load balance:

$$\sum P_G = \sum P_D \tag{7}$$

Node balance (at node i):

$$(P_G)_i + \left(P_{flow_{in}}\right)_i = (P_D)_i + \left(P_{flow_{out}}\right)_i \tag{8}$$

Flow balance (at line j):

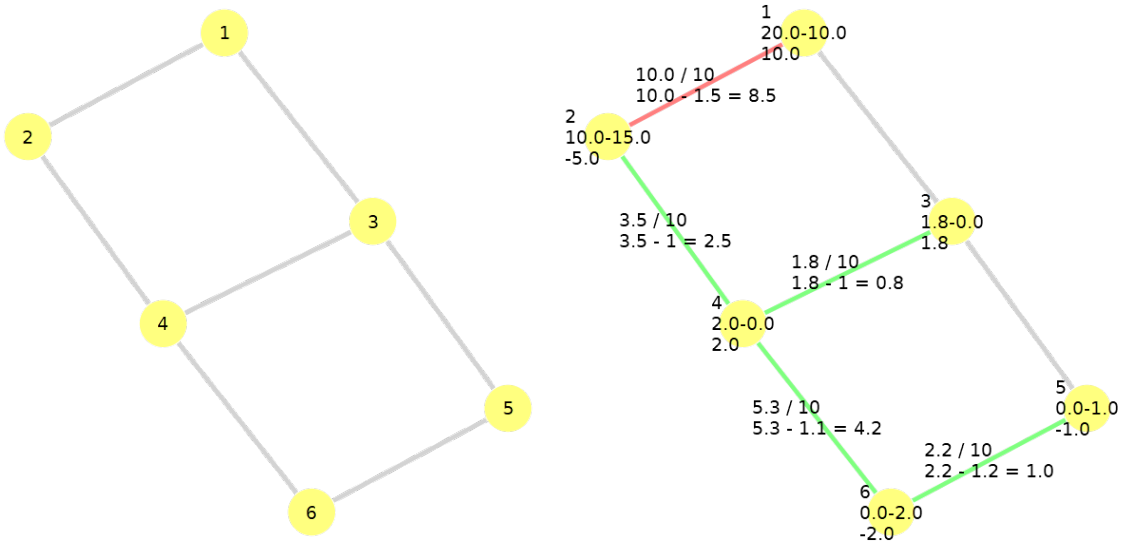
$$\left(P_{flow_{from}}\right)_j = \left(P_{flow_{to}}\right)_j + \left(P_{flow_{loss}}\right)_j \tag{9}$$

With the constraints above in mind, a test configuration is seen in figure 5 to illustrate the success of the model. Figure 5a shows the initial graph with the respective node numbers and 5b shows the graph populated with data from the optimization. The labels represent:

Table 5: Graph labels

Node	**Line**
Node number	flow / capacity
generation - demand	flow from - flow losses = flow to
resulting power generation (negative is demand)	

To elaborate the result shown in figure 5b: Node 1 is demanding 10 units and is generating 20. The surplus 10 units is distributed via line 1-2. The flow loss is 1.5 and only 8.5 reaches node 2. Node 2 is 5 units in deficit and thus absorbs 5 of the 8.5. The remaining 3.5 units are sent to node 4 via line 2-4. Here, the flow loss at this line is 1 and only 2.5 units reaches node 4 and so forth.



(a) Graph configuration **(b)** Optimization result

Figure 5: Test of transport model

The lines in the optimization result are chosen to be colored with respect to the degree of line loading. This will also show up later in the model, which can be found in the github repository [1]. The plot of the graph is done with `GraphPlot.jl` [44].

4.1.4 CO2 emissions

There are 3 approaches to consider CO_2-emission: 1) Direct CO_2-emissions, 2) Direct CO_2-emissions including potent GHG for equivalent CO_2-emissions (like methane), and 3) Lifecycle emissions (including emissions associated from construction and extraction of materials, and potent GHG). The following figure shows the *lifecycle emissions* gathered from IPCC [45, p. 1335]. The energy model in the report uses the median, which equals the mean of the boxplot in the figure.

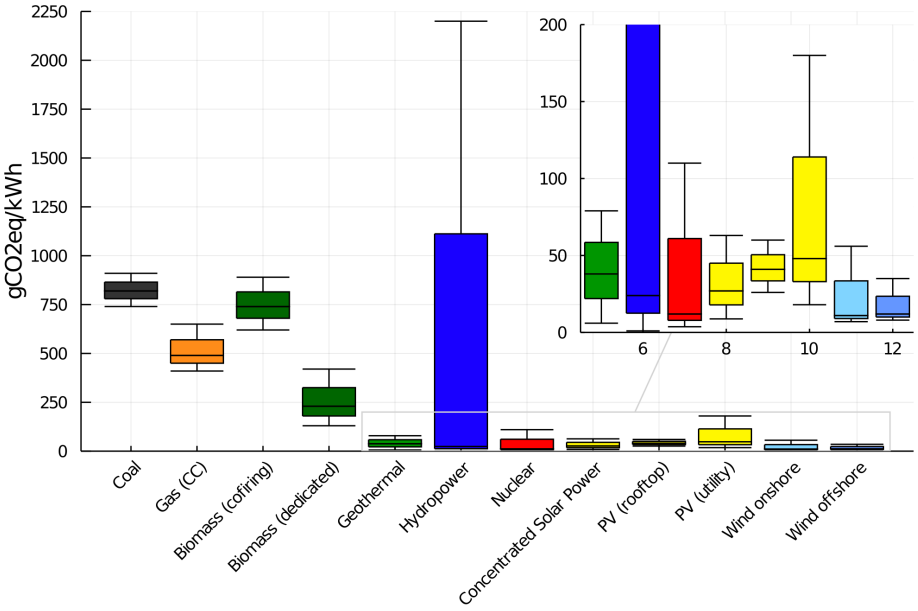


Figure 6: Lifecycle CO2-equivalent emissions

5 Implementing Energy Model in Julia

The implementation of the energy model begins with data acquisition and conditioning with respect to the German scenario. An excel file serves as a database from which the model loads the required input to the UC problem. This database is used create data structures which encompasses information associated with generation capacities, fuel type, federal state, fuel costs etc. Having created information resource, the problem of unit commitment is implemented using different optimization methods and the results are used by the transport model to determine exchange flows among the federal states.

5.1 Model Outline (Unit commitment)

The model considers the power grid in Germany with a "resolution" of 16 states. This is of course not representative for a realistic model but it helps documenting the actual model implementation and corresponding results.

Also, by choosing 16 states, the model can be based on governmental data from *Bundesnetzagentur* (BNA). To create the model, data on generation, capacity, demand and transmission lines are needed. BNA offers via their online platform for the electricity market, "Strommarktdaten für Deutschland" (also known as "SMARD.de"), data on generation and demand. The generation is categorized in energy sources and can be used to simulate variable generation capacities for RES [2]. The demand is used for the demand profile for which the model must dispatch power plants for. The capacities for other energy sources are gathered from BNA's "Kraftwerksliste" [3]. These two data sources categorizes energy sources differently and thus has to be mapped. This is done in the following table:

Table 6: Mapping of data for the energy model

#	smard.de	Kraftwerkliste	Cost €/MWh	η -	Start up €	Emissions gCO2eq/kWh
1	Biomasse	Biomasse	5	1	5000	230
2	Wasserkraft	Laufwasser Speicherwasser (ohne Pumpspeicher)	5	1	5000	24
3	Wind Onshore	Windenergie (Offshore-Anlage)	0	1	0	11
4	Wind Offshore	Windenergie (Onshore-Anlage)	0	1	0	12
5	Photovoltaik	PV	0	1	0	48
6	Sonstige Erneuerbare	Geothermie	0	1	0	38
7	Kernenergie	Kernenergie	5	1	5000	12
8	Braunkohle	Braunkohle	8.9	0.35	5000	820
9	Steinkohle	Steinkohle	8.9	0.35	4500	820
10	Erdgas	Erdgas	16.3	0.5	2500	490
11	Pumpspeicher	Pumpspeicher	5	1	5000	24
12	Sonstige Konventionelle	Abfall Deponiegas Grubengas Klärgas Mehrere Energieträger (nicht erneuerbar) Mineralölprodukte Sonstige Energieträger (nicht erneuerbar)	36.8	0.5	4500	656

To make some assumptions on what the model will output, different parameters are derived and plotted in figure 7. The population distribution describes where to expect high demands. However, the demand

is not based on population but data from SMARD. The sum of capacities per state is plotted to clarify this relation. On that note, the ratio between population and capacity can be plotted as an expression of which states would require incoming power flow.

Since "green" energy sources are cheapest in the model, the share of RES in the capacity (α) and the wind share of RES (β) are considered [46]. The last plot shows total CO_2 emissions if every plant is operated at 100%. These parameters indicates the direction of the power flow when the wind is blowing or sun is shining.

The following are a few sample observations that can be inferred from the figure 7.

- The state of Berlin has a considerable population density while having a low generation capacity, this is reflected as a high demand to capacity of the state.
- The state of Niedersachsen has a moderate population density while has a higher generation capacity, thus has a lower demand to capacity ratio. Further the state has a higher α and β factors which indicates high share of wind energy in the state energy mix. This is further reflected in the states lower potential CO_2 emissions.
- The state of Nordrhein-Westfalen, has a higher generation capacity, while having lower α factor, this indicates a higher generation from conventional power plants and is reflected as a high CO_2 emission potential of the state.

To prepare the data for modelling, state capacity factors are considered to derive the generation capacity of each fuel source for each federal state for a given time. This information serves as an input to the unit commitment problem. This data allows the creation of data structures for power plants corresponding to different federal states and energy source type for simulating the variable capacity of RES.

The cost of fuels for conventional power plants assume recent market prices and a so do the CO_2 emissions [47]. Some of the energy sources such as biomass and nuclear are assumed to have a certain minimum cost. The start up costs are assumptions taken to have a balanced effect on optimization. The RES include no generation costs and penalties. The CO2 emissions per unit generation are taken from IPCC data.

Tables A.2.2, and A.2.3 show the state capacity factors and the derived state capacities with respect to the type of energy source.

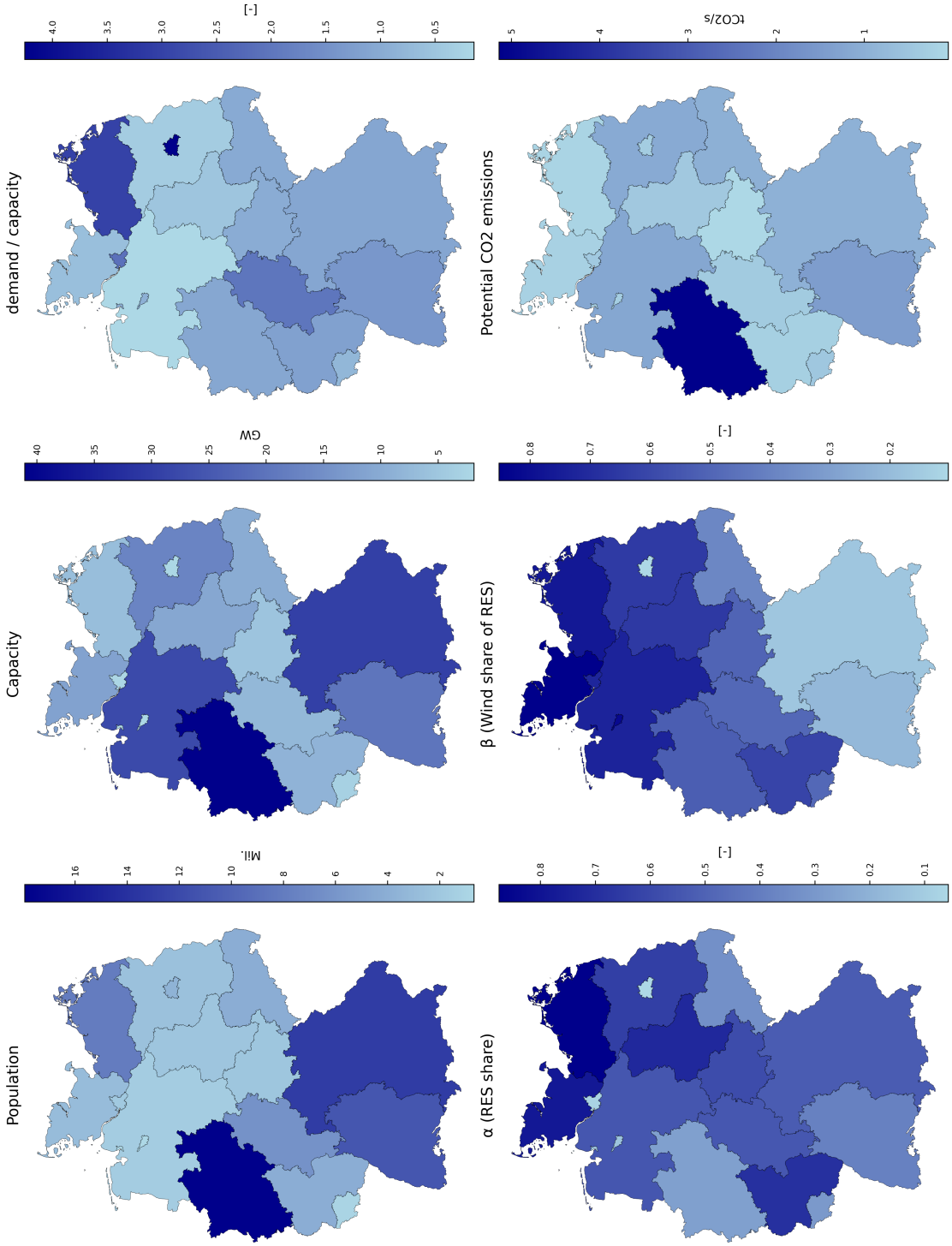


Figure 7: Parameters for Germany

5.2 Technical flow chart

The implemented model in Julia starts by importing relevant packages and loads the relevant and processed data mentioned above.

The data are loaded from excel files into Julia as `DataFrames`. The data is then stored into three different arrays of each `struct` type: `State`, `Plant`, and `Line`. In the specific model, there is $N = 16$ states. The model assumes that each energy fuel is represented by 1 power plant. SMARD categorizes power generation into 12 different categories, and thus does the model assume $N \cdot F = 16 \cdot 12 = 192$ plants representing the power generation of whole Germany.

The model solves and dispatches the power generation at each given time step. This means, in the very basic case, that the model only dispatches the power generation without having any considerations over time. In this way, the model is basically solving a "slice" at a time in the plant matrix, $\mathbf{M}_{\text{plant}} \in \mathbb{R}^{N \times F \times T}$.

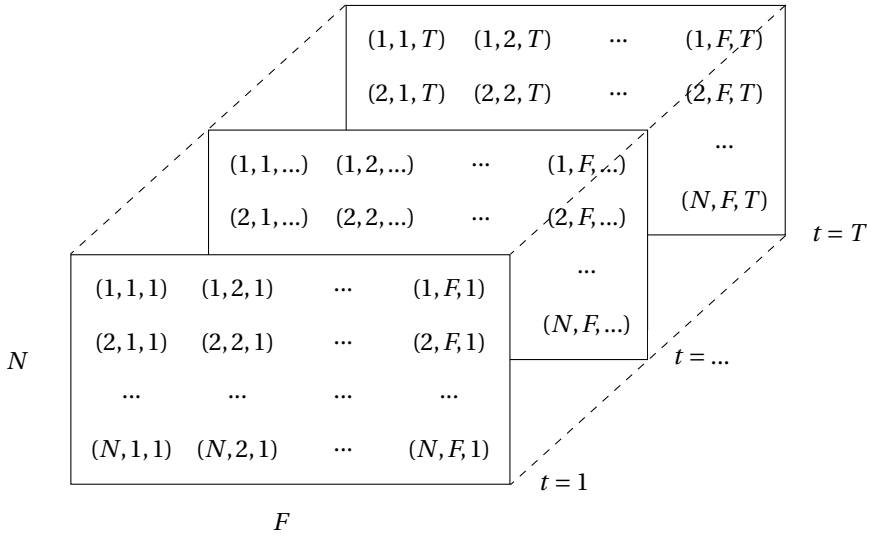


Figure 8: Plant matrix, $\mathbf{M}_{\text{plant}}$

The model can execute different optimization methods based on boolean inputs. The model divides into two branches and a control flow activate one of them based on user input regarding if non-linearity is considered. If the model is linear, several linear constraints are implemented, indicated in the green box in figure 9. The generation constraint is, however, dependent on whether the optimization is including unit activation, x_i. This is indicated by the red box for both the linear and non-linear case.

In the non-linear branch, only the objective function has a non-linear expression. Thus, the only difference of the linear and non-linear branch is that the transport model and the respective constraints are not implemented in the non-linear method.

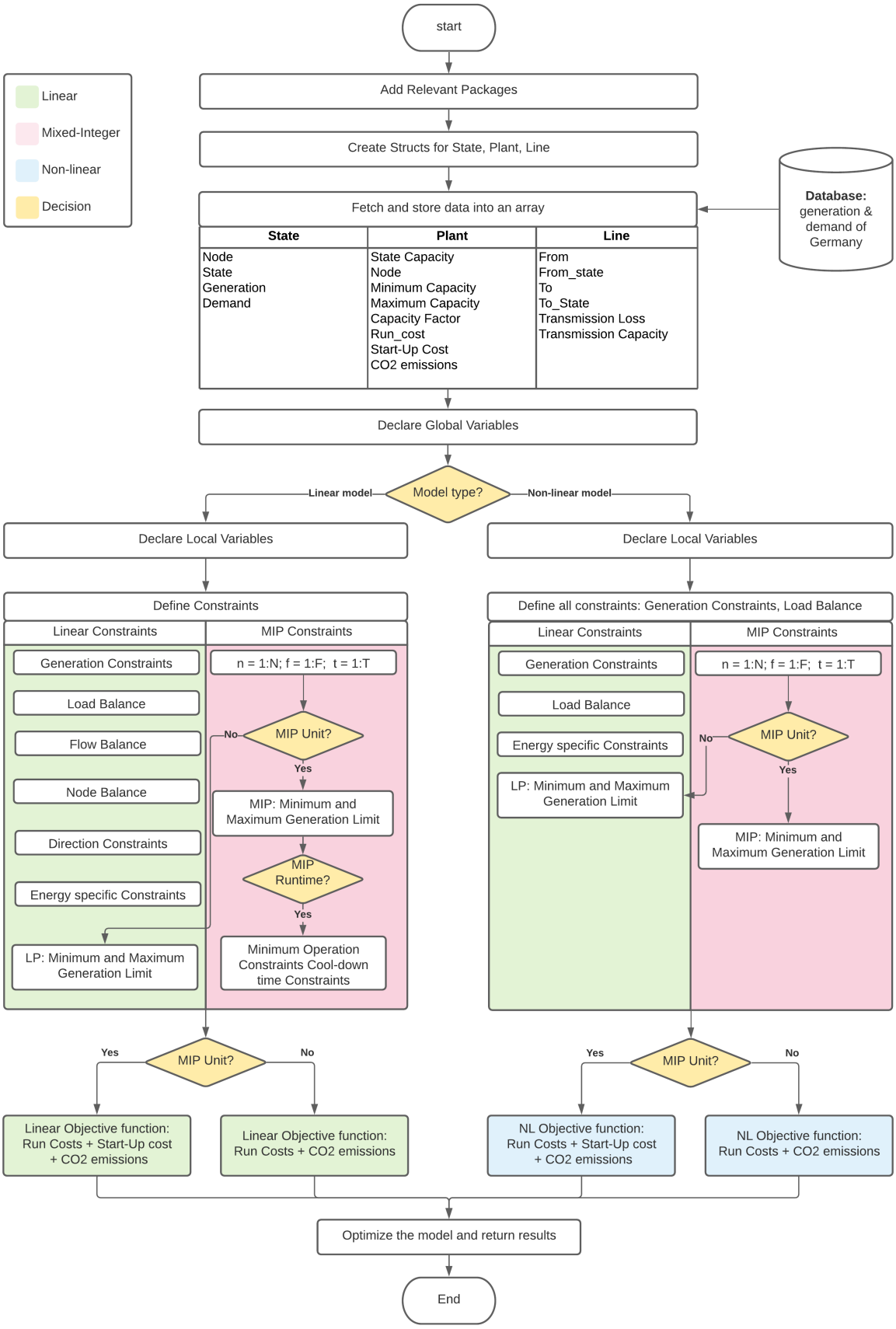


Figure 9: Technical flow chart of implemented energy model in Julia

5.3 Implemented optimization methods

5.3.1 Validation of optimization

Implementation of an energy model with various modelling methods requires validation of these methods independently. In the figure to the right, three main optimization methods are tested and validated, namely LP, MIP, and NLP. Common for all validations is the same parameters for 4 different generators which are modelled with respect to energy cost and capacity. Each model solves a linearly increasing demand.

In the LP-diagram, there is no feasible solution for demands less the sum of than the minimum capacities. When the demand is in the feasible region, the increasing demand is met by a corresponding linear increase of generation from the cheapest generator until the maximum capacity, by which the next cheapest generator starts to increase the generation to meet the increasing demand.

MIP introduces binary variables, which can simulate generators being turned off. This allows the feasible region to start at the lowest value of the minimum capacities. It is noted, that the generation bounces when a minimum capacity has been reached for a cheaper generator due to unit activation.

The NLP is similar to the LP: 1) It has no feasible solutions for values less than the sum of the minimum capacities, 2) it cannot turn off generators, and 3) the NLP follows the same trend of which generators start producing. The difference is the non-linear behavior due to the non-linear partial load function.

As a side note: MOSEK offers a free available *"Modeling Cookbook"* for formulating optimization problems in mathematical terms, which was of great use and advantage. The cookbook was especially helpful for formulating boolean terms regarding the MIP method [48].

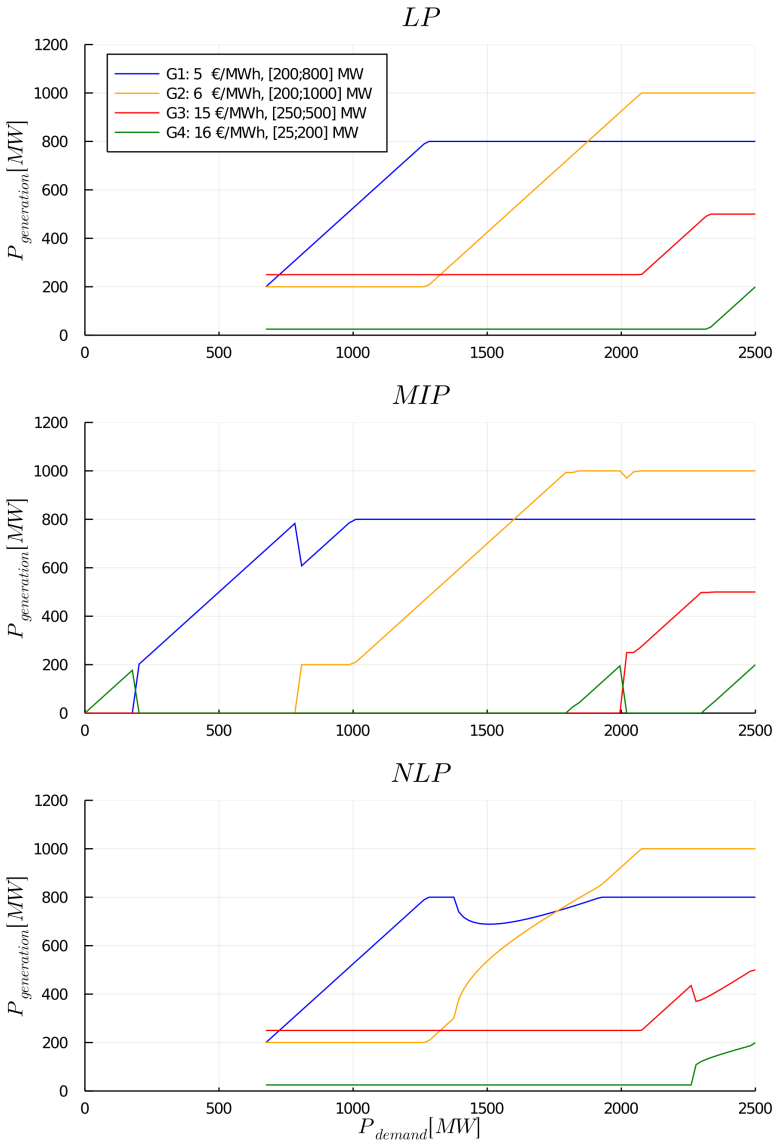


Figure 10: Model validation of optimization methods

5.3.2 Transport Model

The implementation of transport model begins with the extensive simplification of German national transmission system in to a simple system consisting of nodes representing each federal state and their respective interconnections. Figure 11, shows the simplified network to an area resolution of NUTS-1.

The simplification process considers grouping of lines from a given state to another into an equivalent line whose capacity is the sum of individual line capacities, length is equal to the average of line lengths between the two given states and losses proportional to that of individual lines. The simplification proceeds with mapping of network data [49] , for an area resolution of NUTS-2 to an area resolution of NUTS-1, considering state adjacency, number of lines connecting any two given states, their respective capacities and their lengths.

The data corresponding to the number of lines, their capacities and average length can be found in the Tables A.4.2, A.4.1, and A.4.3 respectively available in the appendix section. This data is stored in the database used to create a data structure called "Struct Line" which considers the line terminals (federal states), capacity and losses. This information, available in the data structure, is used in the optimization process for the computation of interstate power flows in parallel with optimized generation. Table A.3 in the appendix section shows the information in data structure "Struct Line".

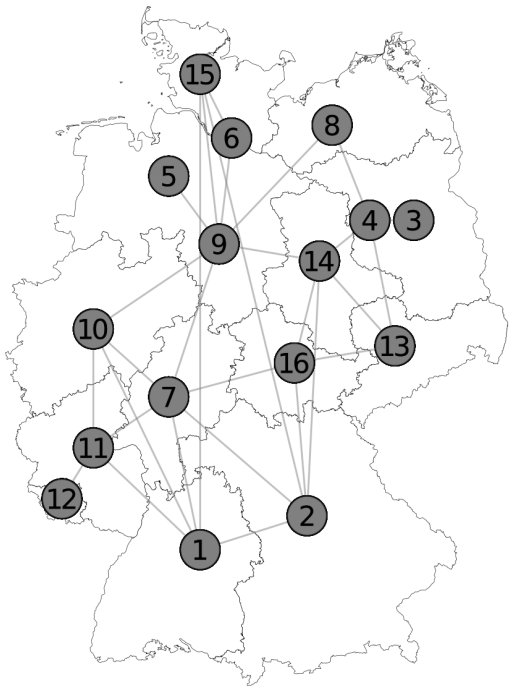
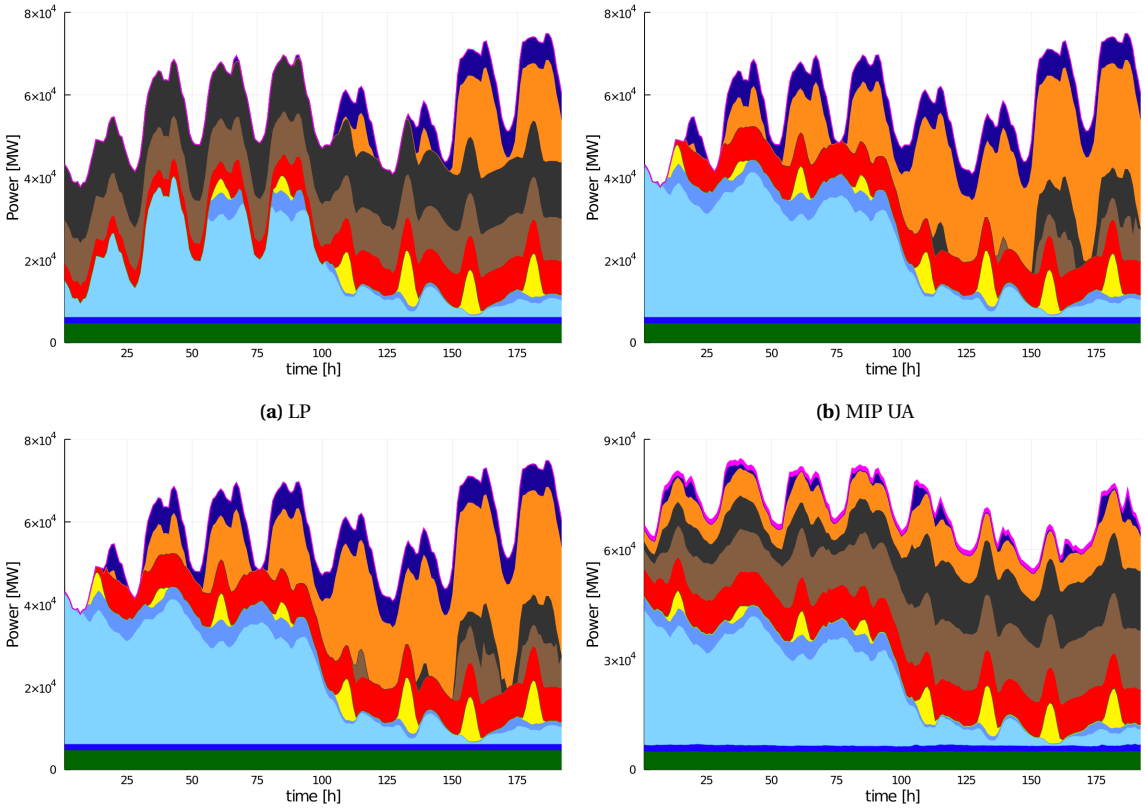


Figure 11: Interstate transmission lines

6 Observations and Results

6.1 Comparison of programming methods

Energy modelling plays a vital role in providing an optimized solution for the energy demand of a specific area. However, it becomes crucial to decide what factors and constraints should be considered during the optimization process of any energy model. The below plots were obtained through three different optimization methods in Julia, i.e. 1) LP, 2) MIP with Unit Activation, and 3) MIP with Run-time constraints. These are plotted against real data for a summer and a winter scenario. The following figure shows the winter scenario:



(a) LP

(b) MIP UA

(c) MIP RT

(d) Real data

Figure 13: Optimization results for winter case (2019/01/14 - 2019/01/21)

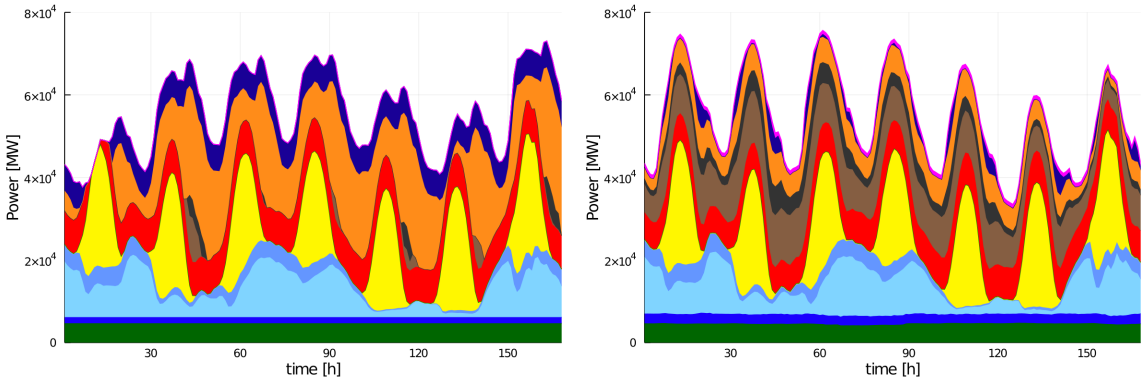
The LP method is more comparable to the real data, but this method does not take into account the unit activation and shut-down of coal power plants. As a result of this, there is a reduced generation by renewable sources in the LP method.

On the other hand, two different MIP methods are implemented taking into consideration the unit activation and shut-down time of the lignite and coal power plants. As a result of this consideration, the optimized solution obtained by these two methods include a minimal production from lignite and coal power plants. Although, the output of these two methods resembles each other with only a slight differences in generation by lignite and coal power plants, these methods largely differ in the computation time, with RT taking more time for computation as compared to the UA method. Moreover, the production by the natural gas are much higher in case of MIP as compared to the original data due to the fact that the CO_2 emission cost of natural gas are much cheaper than that of the lignite or hard coal power plants.

Figure 12: Legend for figures 13 & 14

A major difference can also be observed in the net real generation and the net modelled generation. This is due to the fact that the modelled generation does not take into account the additional generation for spinning reserves and import/export between various territories.

Figure 14 represents an optimized energy model in Germany for the summer. It should be noted that the total generation during the summer is less than in winter due to the less energy demand in summer. Furthermore, there is clearly an upsurge in the energy production by solar PVs during summer, although a plunge can be noted in the wind energy generation. Despite this drop, there is fairly a large variation between the power generated by hard coal and lignite power plants in the two instances due to the high penalty costs for CO_2 emissions. Interestingly, due to the decreased generation from the coal power plants, the pump storage units rockets to some significant percentage of generation in the modelled data, although these are relatively low in the real case scenario.



(a) MIP UA

(b) Real data

Figure 14: Optimization results for summer case (2019/06/24 - 2019/06/30)

6.2 Congestion in the transmission system

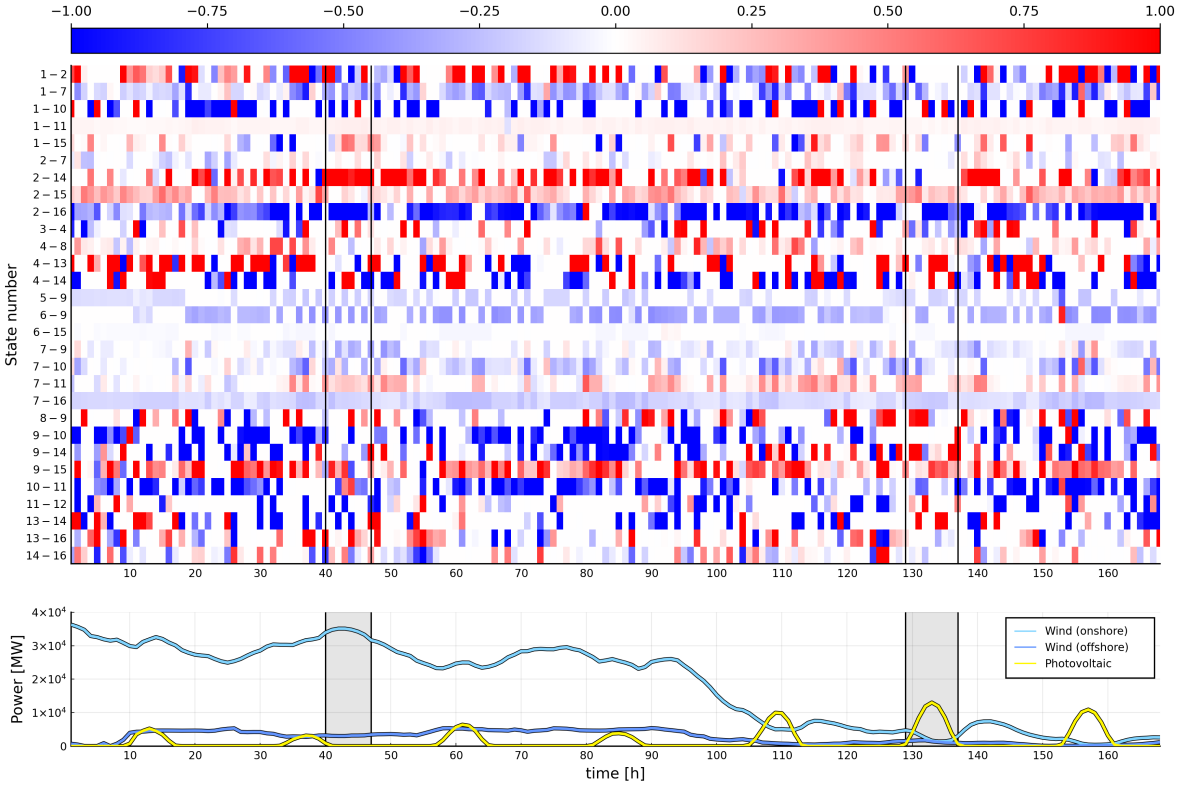
The energy model incorporates a transmission system and congestion might be a critical factor for distributing the "cheapest" and "greenest" power. In this section it will be analyzed if the energy model points out congestion issues due to large amount of RES. In figure 15, two plots are given for documenting the loads of the transmission lines in per unit for a winter and a summer case. In both plots, a heatmap and a lineplot is given. The y-axis of the heatmap is mapping the transmission line between the states[6] with the time step on the x-axis. Below the heatmap, a lineplot shows the power generation of wind and PV. The two grey areas have been added to focus the periods with either dominating wind or PV generation for the respective time of the year.

The assumption is that congestion issues from high RES shares are indicated by systematic, continuous periods with a constant, saturated color. The different colors tell the direction of the flow: If the color is red, the flow is going from $a \rightarrow b$. If blue, the flow is going $a \leftarrow b$. The loads of the lines in figure 15 requires further elaboration[7]:

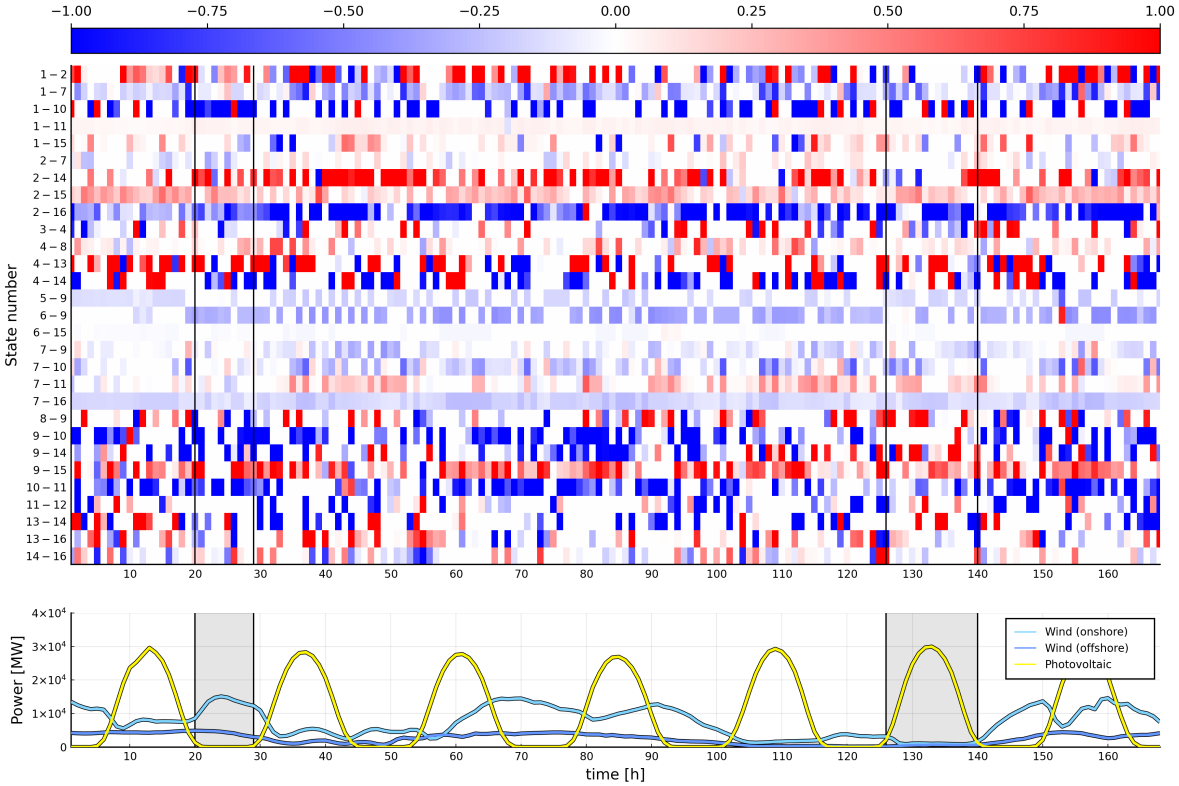
In the first marked area in figure 15a, there is a high power generation from wind, and only 2 lines appear to be fully loaded in that period, namely line $2 \rightarrow 14$ and $2 \leftarrow 16$. In appendix A.2.1, it is seen that state 2 (Bayern) has a low share of wind (β) which counters the assumption of RES being the cause of congestion in this case. However, state 16, Sachsen-Anhalt, has a high share of RES and wind (α, β) and a low transmission capacity to state 2 (2 GW). This indicates a congestion from RES. But this behaviour does not seem to be systematic and cannot be explained due to local weather conditions because these are not included in the model.

[6] Represented by numbers. The numbers can be further mapped in figure 11 and in appendix A.3

[7] When referring to the lines, the general flow for the line can be assigned to the reference, E.g. line $1-2$ can be refered to as $1 \rightarrow 2$ if the general tendency of the flow is going from state 1 to 2.



(a) Winter case: 2019/01/14 - 2019/01/21



(b) Summer case: 2019/06/24 - 2019/06/30

Figure 15: Transmission line loading over time

In comparison to the second marked period with relative high PV generation for the time of the year, line $2 \leftarrow 16$ is congested when the PV generation peaks. But this is not systematic with respect to the rest of the graph and thus fail to prove a congestion pattern from RES.

Regarding the summer case in figure 15b: In the first marked period dominated by wind generation, line $1 \leftarrow 10$ seems continuous regarding the magnitude of the load but shifts direction at 1 time step. This does not seem systematic for the rest of the plot. Also, the RES and wind share (α, β), shown in appendix A.2.1, imply that the congestion is not mainly related to RES.

In the second marked period, nothing seems to be continuous or to congest when the power generation from PV peaks. However, line $7 \leftarrow 16$ is peaking with the PV generation. The line is not congested even though it is only have a capacity of 3.35 GW. If that line is investigated for systematic behavior, there can be seen load peaks "in phase" with PV generation. This is also valid for the winter case. The systematic behavior is, however, far from pointing to a congestion issue.

It seems difficult for the energy model to point out congestion patterns related to RES at first hand. An explanation could be that there is not simulated any losses at the lines[8]. This means that the power virtually can flow through many states before being consumed at the demanding state. This also implies that direct transmission lines with a specific purpose of dealing with RES, like line $1-15$ and $2-15$, do not necessarily congest even though there is a high share of RES in the energy mix at certain times.

6.3 Solution time

Not all aspects and constraints improve the result significantly regarding the additional computation time. Solving times has been bench-marked for the different modelling approaches with following system specifications:

Table 7: System specifications

Processor	Intel(R) Core(TM) i7-7500U CPU @2.70GHz 2.90 GHz
Installed RAM	8.00 GB (7.89 GB usable)

The benchmark in the figure to the right is showing the average solution time of 10 solutions for x number of time steps for the various modelling options[9] featured in the program.

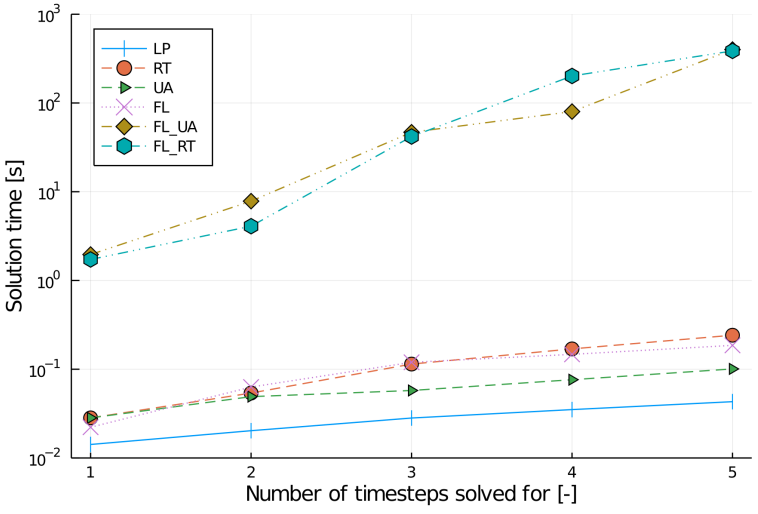


Figure 16: Solution time

Speaking of optimization method with respect to the figure: The solid line is LP, dashed lines are MIP, dotted lines are LP with flow losses, and dashed-dotted lines are MIP with flow losses. The significantly increase in solution time is seen with a combination of MIP and flow losses.

A NL objective function was implemented with success. However, NLP and MINLP are solved with additional model parameters such as 1) time limit for branch solution, and 2) whether the model accepts "almost locally solved" solutions. The latter means, given low gradients in model, that the solution cannot be categorized as the optimal solution.

[8] This is because the losses can only be constant to keep linearity in the model. And even if constant transmission losses are included, the computation time will rapidly increase with dubious modelling result of actual transmission losses as function of transmitted power.

[9] LP: Linear programming. RT: MIP included to calculate runtime costs. UA: MIP including minimum unit activation and down time. FL: Flow losses included, which add an extra nodal analysis for each line

Therefore, the NLP and MINLP has not been plotted along since it would require specifications beyond the scope of this report.

7 Conclusion

7.1 Summary

The objective of modelling the *unit commitment* in Julia was successful. Based on the clarification of the energy model approach in section 4, the sections 5 and 6 showed the results of the optimized model implemented in Julia with the optmization package, `JuMP`. The implementation included various optimization methods as tabulated in Table 8:

Table 8: Implemented optimization methods

LP	QP	SOCP	MILP	NLP	MINLP	SDP
✓			✓	✓	✓	

With the many different solvers, and the possibility to customize the model with multiple solvers for each kind of optimization problem, the JuMP package is a very strong foundation to solve optimization problems in general.

Plotting packages for visualizing results are available to a high degree but there is still a lot features missing and some features are buggy if more customized plots are demanded. Almost every plot in the report is based on `Plots` package.

This energy model optimized the power generation in Germany by utilizing the historic demand data of 2019 for both summer and winter, at the lowest cost. The minimization of the cost of power generation included summing up of fuel costs, CO_2 emissions costs and the start up costs for conventional power plants.

However, the fuel costs of various power plants were based on assumptions and approximations.

Besides cost assumptions, only few out of many constraints were incorporated in this model. The constraints like ramp rates and spinning reserves were not included in the implementation. This resulted in an energy mix with too little power generation from coal and lignite power plants, although the power generation from RES was comparable to the reality. The probable reasons for this deviation could be accredited to the high penalizing costs for life cycle CO_2 emissions from the conventional power plants. A more close and accurate result would have been possible had this energy model been successful in implementing the transport and NLP model together.

Furthermore, the transport model was implemented successfully with MILP model along with investigating the typical congestion issues due to high RES share in the energy mix. However, no clear congestion issues could be addressed. This could be because the actual transmission losses were not included as it overwhelmingly increased the solution time when combining flow losses with MIP modelling (which is necessary for the best result regarding RES shares in the energy mix).

It is hoped that this showcase can contribute to further engagement. The code, documentation, and data for the model can be found on Github [1].

7.2 Discussion

It is evident from the above sections that Julia is a promising language for scientific computation. It is a blend of some unique and capable features such as high-level high performance language, dynamic programming language, syntax similar to that used in scientific computation environments and an extensive mathematical library. Julia has the best libraries of any language for differential equations, mathematical optimization and automatic differentiation. Further, Julia has a general core and extends beyond being just a language for doing math. Julia provides general abstractions on networking, concurrency, process orchestration, text parsing, interacting with the host OS and calling C.

However some issues are of concern with the language. The language while being high performance has some lags and requires some time for the JIT (Just in Time) to warm up. Furthermore, as third party pack-

ages are loaded the warmup time becomes substantial. The same is true with time required to develop plots. Also the interface to native APIs is not safe. The language has a relatively unfamiliar and complicated syntax and its current implementation limits flexibility. The ecosystem is not general and favours scientific niche.

The language is yet to achieve some benchmarks for which there are several ongoing discussions on the different platforms and communities.

7.3 Future outlooks

It is worth noting that Julia is still a young language, but it's capabilities are well competent with the existing languages like Python, R and C. There are a large collection of Julia packages for multiple purposes, and in addition, several new packages would be added as the Julia community evolves with increased participation and developers. The extent of development in packages is reflected from the availability of specific package such as `PowerModels.jl`, which is a steady state power network optimizer. The community behind PowerModels also calls for collaboration [50], which is an expression of the general collaborative approach, which has been experienced through several engagements with the Julia community.

The expectation of the scientific community from the language is growing with its evolution. Several organizations such as N26, DataPipeline, Platform Project, CavalRe, Chai, and Flitto are reported using Julia in their tech stacks and popular tools Plotly.js, Octave, AnyChart, XGBoost, and MXNet are reported to be integrated with Julia. It is therefore imperative to anticipate the increasing prominence of Julia in the near future.

A Tables

A.1 Plant correction factors

Power Plant Type	Acronym	a	b	P_{ratio}	D_1	D_2	S_2	S_1	S_0
Gas turbine	GT	-0.052	-0.602	1.00	0.1287	-0.3183	-0.000013	-0.00039	1
Steam turbine	ST	0.051	-0.611	0.85	0.0288	0.2188	-0.0000087	-0.00104	1
Combined cycle	CC	0.080	-0.750	0.98	0.0065	0.4230	-0.000013	-0.00039	1
IC engines	IC	-0.455	0.272	1.00	0.0160	0.4083	-0.0000087	-0.00104	1

A.2 State data

A.2.1 Central figures

#	State	NUTS1	Pop. [mil.]	Pop. [%]	Cap. [GW]	Cap. [%]	$\frac{\text{Demand}}{\text{Capacity}}$ [-]	α [%]	β [%]	Pot. emis. $\frac{tCO_2}{s}$
1	Baden-Württemberg	DE1	11.10	13.3%	19.830	9.7	1.380	38.6	21.2	1.445
2	Bayern	DE2	13.12	15.8	29.500	14.4	1.097	52.7	16.4	0.966
3	Berlin	DE3	3.67	4.4	2.127	1.0	4.254	5.7	10.3	0.300
4	Brandenburg	DE4	2.52	3.0	17.137	8.4	0.363	63.8	65.0	1.128
5	Bremen	DE5	0.68	0.8	1.859	0.9	0.903	13.3	81.9	0.285
6	Hamburg	DE6	1.85	2.2	2.172	1.1	2.096	7.6	72.1	0.400
7	Hessen	DE7	6.29	7.6	7.702	3.8	2.013	54.4	49.2	0.366
8	Mecklenburg-Vorpommern	DE8	7.99	9.6	6.556	3.2	3.005	87.6	76.8	0.149
9	Niedersachsen	DE9	1.61	1.9	27.632	13.5	0.144	54.9	72.9	1.211
10	Nordrhein-Westfalen	DEA	17.93	21.6	41.063	20.0	1.077	27.0	52.8	5.142
11	Rheinland-Pfalz	DEB	4.08	4.9	8.496	4.1	1.185	69.4	61.6	0.248
12	Saarland	DEC	0.99	1.2	3.152	1.5	0.775	29.5	49.2	0.446
13	Sachsen	DED	4.08	4.9	9.895	4.8	1.016	33.0	38.3	0.994
14	Sachsen-Anhalt	DEE	2.21	2.7	10.793	5.3	0.504	72.5	65.4	0.415
15	Schleswig-Holstein	DEF	2.90	3.5	11.690	5.7	0.611	78.5	85.2	0.206
16	Thüringen	DEG	2.14	2.6	5.417	2.6	0.975	58.6	50.6	0.048

A.2.2 Capacity Factors

The values are mapped from Kraftwerklist into SMARD generation categories and then the relative capacity factor is found.

Abbreviations:

Bio	Biomasse
Hydro	Wasserkraft
Won	Wind onshore
Woff	Wind offshore
PV	Photovoltaik
Geo	Sonstige Erneuerbare
N	Kernenergie
L	Braunkohle
HC	Steinkohle
NG	Erdgas
PS	Pumpspeicher
OthC	Sonstige Konventionelle

[%]	Bio	Hydro	Won	Woff	PV	Geo	N	L	HC	NG	PS	OthC
DE1	11.73	18.01	3.09		12.75	1.32	16.14		24.35	3.90	29.46	9.51
DE2	22.10	58.83	4.83		27.48	79.91	33.25		3.74	15.88	8.54	19.11
DE3	0.52		0.02		0.23				2.89	4.00		2.88
DE4	5.51	0.13	13.49		8.10			20.94		2.96		6.81
DE5	0.15	0.27	0.39		0.09				3.41	1.74		4.05
DE6	0.49		0.23		0.10				7.93	0.57		0.27
DE7	3.35	2.26	3.91		4.50			0.16	3.33	5.86	9.83	2.19
DE8	4.30	0.08	6.31	50.00	4.24				2.27	1.21		0.46
DE9	20.06	1.50	21.00		8.69		33.23	1.69	12.97	15.53	3.46	5.19
DEA	10.63	4.70	11.08		11.06			51.15	29.41	31.67	4.77	33.70
DEB	2.16	6.37	6.89		4.79	18.77			0.06	7.42		2.36
DEC	0.24	0.31	0.87		1.00				8.06	0.59		2.42
DED	3.29	5.77	2.37		4.25			20.75		2.67	17.07	0.46
DEE	5.67	0.76	9.71		5.73			5.30		3.18	1.25	5.10
DEF	6.74	0.13	12.75	50.00	3.69		17.38		1.58	1.20	1.87	5.28
DEG	3.06	0.88	3.05		3.31					1.64	23.74	0.19

A.2.3 Capacity (Kraftwerklist)

Abbreviations:

Abf	Abfall
Dgas	Deponiegas
Kgas	Klärgas
Ggas	Grubengas
SConv	Sonstige Energieträger (nicht erneuerbar)
MConv	Mehrere Energieträger (nicht erneuerbar)
Oil	Mineralölprodukte
NG	Erdgas
Bio	Biomasse
L	Steinkohle
HC	Abfall
Geo	Geothermie
N	Kernenergie
PV	Photovoltaik
PS	Pumpspeicher
SHyd	Speicherwasser (ohne Pumpspeicher)
LHyd	Laufwasser
Won	Wind onshore
Woff	Wind offshore

[MW]	Abf	Dgas	Kgas	Ggas	SConv	MConv	Oil	NG	Bio	L	HC	Geo	N	PV	PS	SHyd	LHyd	Won	Woff
DE1	94	15	19		10		702	1029	961		5506	1	1310	6030	1873		656	1625	
DE2	214	11	16		58		1388	4196	1810		847	33	2698	13000	543	170	1973	2546	
DE3	36						218	1056	43		653			108				12	
DE4	118	24	2		125		334	781	451	4364				3830			5	7104	
DE5	91	2			178		86	459	12		772			45			10	203	
DE6	24							150	40		1794			46				119	
DE7	112	19	10		28		25	1548	275	34	753			2129	625	20	62	2062	
DE8	17	9	2		13			319	352		514			2005			3	3323	3324
DE9	73	13	8		289	19	56	4103	1644	352	2933		2696	4108	220		55	11063	
DEA	517	36	21	130	1557	169	545	8367	871	10658	6650			5229	303	15	156	5839	
DEB	102	8	1		98			1959	177		13	8		2268			232	3631	
DEC	28			56	130			155	20		1822			473			11	457	
DED	16	7					17	705	270	4325				2010	1085		210	1250	
DEE	183	11	1		43		213	841	465	1104				2708	80		28	5117	
DEF	33	7	3		146		276	317	553		357		1410	1745	119		5	6719	3324
DEG	12	4	1					432	251					1567	1509		32	1608	

A.3 Structs

The model is constructed from 3 different structs: `Line`, `State`, and `Plant`. Because `State` and `Plant` are time dependent, the specific tables will not be brought here but can be found in the repository [1].

from	to	from_state	to_state	loss [MW]	capacity [MW]
1	2	Baden-Württemberg	Bayern	4.31	10625
1	7	Baden-Württemberg	Hessen	3.8016	6225
1	10	Baden-Württemberg	Nordrhein-Westfalen	3.136	2000
1	11	Baden-Württemberg	Rheinland-Pfalz	0.5865	4550
1	15	Baden-Württemberg	Schleswig-Holstein	6.272	2000
2	7	Bayern	Hessen	1.7919	9000
2	14	Bayern	Sachsen-Anhalt	4.128	2000
2	15	Bayern	Schleswig-Holstein	4.77	2000
2	16	Bayern	Thüringen	2.3296	3350
3	4	Berlin	Brandenburg	1.634	11200
4	8	Brandenburg	Mecklenburg-Vorpommern	1.3432	9000
4	13	Brandenburg	Sachsen	3.0016	10050
4	14	Brandenburg	Sachsen-Anhalt	3.8732	14550
5	9	Bremen	Niedersachsen	4.2098	14750
6	9	Hamburg	Niedersachsen	1.2462	3350
6	15	Hamburg	Schleswig-Holstein	1.1997	6700
7	9	Hessen	Niedersachsen	2.1522	1150
7	10	Hessen	Nordrhein-Westfalen	2.6562	2250
7	11	Hessen	Rheinland-Pfalz	2.7144	15225
7	16	Hessen	Thüringen	2.8662	3350
8	9	Mecklenburg-Vorpommern	Niedersachsen	0.9752	3350
9	10	Niedersachsen	Nordrhein-Westfalen	6.468	14975
9	14	Niedersachsen	Sachsen-Anhalt	1.512	3350
9	15	Niedersachsen	Schleswig-Holstein	3.3176	11200
10	11	Nordrhein-Westfalen	Rheinland-Pfalz	3.0758	10775
11	12	Rheinland-Pfalz	Saarland	2.95	9000
13	14	Sachsen	Sachsen-Anhalt	3.0912	575
13	16	Sachsen	Thüringen	5.0601	7850
14	16	Sachsen-Anhalt	Thüringen	1.0692	2300

A.4 Transmission Lines

All the transmission lines are mapped via NUTS level 2 to 1. Values in the diagonal are results from the mapping and can be understood as intra-state capacities, number of transmission lines, and the average length.

A.4.1 Total transmission capacities

[MW]	DE1	DE2	DE3	DE4	DE5	DE6	DE7	DE8	DE9	DEA	DEB	DEC	DED	DEE	DEF	DEG
DE1	37625	10625					6225			2000	4550				2000	
DE2	10625	29300					9000							2000	2000	3350
DE3				11200												
DE4			11200	7850				9000					10050	14550		
DE5									14750							
DE6									3350						6700	
DE7	6225	9000					11200		1150	2250	15225					3350
DE8				9000					3350							
DE9					14750	3350	1150	3350	20350	14975				3350	11200	
DEA	2000						2250		14975	73350	10775					
DEB	4550						15225			10775	2300	9000				
DEC											9000					
DED				10050									10625	575		7850
DEE		2000		14550					3350				575			2300
DEF	2000	2000				6700			11200							
DEG		3350					3350						7850	2300		

A.4.2 Number of lines

[-]	DE1	DE2	DE3	DE4	DE5	DE6	DE7	DE8	DE9	DEA	DEB	DEC	DED	DEE	DEF	DEG
DE1	4	3					1			1	1				1	
DE2	3	10					1							1	1	1
DE3				2												
DE4			2	1				1					1	2		
DE5									3							
DE6									1						1	
DE7	1	1					3		1	2	3					1
DE8				1					1							
DE9					3	1	1	1	4	4				1	1	
DEA	1						2		4	6	2					
DEB	1						3			2	2	2				
DEC											2					
DED				1									3	1		2
DEE		1		2					1				1			1
DEF	1	1				1			1							
DEG		1					1						2	1		

A.4.3 Average Length

[km]	DE1	DE2	DE3	DE4	DE5	DE6	DE7	DE8	DE9	DEA	DEB	DEC	DED	DEE	DEF	DEG
DE1	107.25	143.6667					352			320	51				560	
DE2	143.6	120.9					181							430	450	208
DE3				86												
DE4			86	369				146					268	210.5		
DE5									144.6							
DE6									134						129	
DE7	352	181					110		211	116.5	77.33333					281
DE8				146					92							
DE9					144.6	134	211	92	78.5	147				140	286	
DEA	320						116.5		147	118.16	169					
DEB	51						77.3			169	146	147.5				
DEC											147.5					
DED				268									138.3	276		250.5
DEE		430		210.5					140				276			108
DEF	560	450				129			286							
DEG		208					281						250.5	108		

References

[1] PROPENS, "Project github repository." https://github.com/bvilmann/showcase_jl_energy_modelling. Accessed: 2021-01-24.

[2] Bundesnetzagentur, "Smard.de." https://www.smard.de/en/downloadcenter/download-market-data#!?downloadAttributes=%7B%22selectedCategory%22:1,%22selectedSubCategory%22:1,%22selectedRegion%22:%22DE%22,%22from%22:1610578800000,%22to%22:1611529199999,%22selectedFileType%22:%22XLS%22%7D. Accessed: 2021-01-24.

[3] Bundesnetzagentur, "Kraftwerkliste." https://www.bundesnetzagentur.de/DE/Sachgebiete/ElektrizitaetundGas/Unternehmen_Institutionen/Versorgungssicherheit/Erzeugungskapazitaeten/Kraftwerksliste/kraftwerksliste-node.html. Accessed: 2021-01-24.

[4] M. Lubin and I. Dunning, "Computing in operations research using julia," Informs Journal on Computing, vol. 27, no. 2, pp. 238–248, 2015.

[5] Tiobe, "The tiobe index." https://www.tiobe.com/tiobe-index. Accessed: 2021-01-23.

[6] Julia, "Newsletter january 2021." https://juliacomputing.com/blog/2021/01/newsletter-january/. Accessed: 2021-01-23.

[7] Julia, "Noteworthy differences from other languages." https://docs.julialang.org/en/v1/manual/noteworthy-differences/. Accessed: 2021-01-23.

[8] JuMP, "Documentation: Pkg (functions)." https://docs.julialang.org/en/v1/manual/functions/#Varargs-Functions. Accessed: 2021-01-23.

[9] JuMP, "Documentation: Frequently asked questions." https://docs.julialang.org/en/v1/manual/faq/#The-two-uses-of-the-...-operator:-slurping-and-splatting. Accessed: 2021-01-23.

[10] GitHub, "Documentation: Pipe." https://github.com/sylvaticus/juliatutorial/blob/master/useful-packages/pipe.md.

[11] Julia, "Documentation: Metaprogramming." https://docs.julialang.org/en/v1/manual/metaprogramming/. Accessed: 2021-01-23.

[12] stackoverflow, "Documentation: what-is-a-symbol-in-julia." https://stackoverflow.com/questions/23480722/what-is-a-symbol-in-julia. Accessed: 2021-01-23.

[13] Julia, "Documentation: Argument passing behavior." https://docs.julialang.org/en/v1/manual/functions/#Argument-Passing-Behavior. Accessed: 2021-01-23.

[14] Stackoverflow, "What does an exclamation mark mean after the name of a function?." https://stackoverflow.com/a/45397024/10156184. Accessed: 2021-01-23.

[15] Discourse, "How to warn new users away from metaprogramming." https://discourse.julialang.org/t/how-to-warn-new-users-away-from-metaprogramming/35022. Accessed: 2021-01-23.

[16] Julia, "Documentation: Pkg (package handler)." https://docs.julialang.org/en/v1/stdlib/Pkg/. Accessed: 2021-01-23.

[17] github, "Juliasmoothsolvers." https://github.com/JuliaSmoothOptimizers. Accessed: 2021-01-24.

[18] github, "Julianlsolvers." https://github.com/JuliaNLSolvers. Accessed: 2021-01-24.

[19] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," Siam Review, vol. 59, no. 2, pp. 295–320, 2017.

[20] github, "Convex." https://github.com/jump-dev/Convex.jl. Accessed: 2021-01-24.

[21] JuMP, "Documentation: Dataframes.jl." https://dataframes.juliadata.org/stable/. Accessed: 2021-01-23.

[22] GitHub, "Documentation: Excelreaders." https://github.com/queryverse/ExcelReaders.jl.

[23] GitHub, "Documentation: Excelfiles.jl." https://juliapackages.com/p/excelfiles.

[24] GitHub, "Documentation: Fileio.jl." https://github.com/JuliaIO/FileIO.jl.

[25] Plots, "Plots.jl." https://github.com/JuliaPlots/Plots.jl. Accessed: 2021-01-23.

[26] Makie, "Makie.jl." https://github.com/JuliaPlots/Makie.jl. Accessed: 2021-01-23.

[27] Plots, "Input data, columns are series." http://docs.juliaplots.org/latest/input_data/#columns-are-series. Accessed: 2021-01-23.

[28] Plots, "Plots gallery." https://goropikari.github.io/PlotsGallery.jl/. Accessed: 2021-01-23.

[29] Plots, "Backends." http://docs.juliaplots.org/latest/backends/. Accessed: 2021-01-23.

[30] Makie, "Layout tutorial." https://makie.juliaplots.org/dev/makielayout/tutorial.html. Accessed: 2021-01-23.

[31] Makie, "Makie gallery." http://juliaplots.org/MakieReferenceImages/gallery/. Accessed: 2021-01-23.

[32] Stackoverflow, "Julia (jump): Indicator constraints with multiple conditional values (is a boolean expression possible?)." https://stackoverflow.com/questions/65072615/julia-jump-indicator-constraints-with-multiple-conditional-values-is-a-boole. Accessed: 2021-01-24.

[33] Stackoverflow, "Julia: How to introduce binary integers for mixed integers optimization problems with jump?." https://stackoverflow.com/questions/64967656/julia-how-to-introduce-binary-integers-for-mixed-integers-optimization-problems. Accessed: 2021-01-24.

[34] Stackoverflow, "Julia: How can i align x axises in grid plot with plots.jl?." https://stackoverflow.com/questions/65991604/julia-how-can-i-align-x-axises-in-grid-plot-with-plots-jl. Accessed: 2021-01-24.

[35] Discourse, "Binary integer variable returned as float64." https://discourse.julialang.org/t/binary-integer-variable-returned-as-float64/50647. Accessed: 2021-01-24.

[36] Discourse, "Constraining binary integer variables with indicator constraints." https://discourse.julialang.org/t/constraining-binary-integer-variables-with-indicator-constraints/51043. Accessed: 2021-01-24.

[37] Discourse, "How to get started to contribute to julia and packages." https://discourse.julialang.org/t/how-to-get-started-to-contribute-to-julia-and-packages/51489. Accessed: 2021-01-24.

[38] Github, "[bug] colors are not assigned to pie chart properly #3214." https://github.com/JuliaPlots/Plots.jl/issues/3214. Accessed: 2021-01-24.

[39] Github, "[bug] pyplot backend ignores plot order #3196." https://github.com/JuliaPlots/Plots.jl/issues/3196. Accessed: 2021-01-24.

[40] Github, "Some improvements of pie charts #782." https://github.com/JuliaPlots/Makie.jl/issues/782. Accessed: 2021-01-24.

[41] Github, "Plotting mixed graphs #134." https://github.com/JuliaGraphs/GraphPlot.jl/issues/134. Accessed: 2021-01-24.

[42] P. Bihari, G. Gróf, and I. Gács, "Efficiency and cost modelling of thermal power plants," Thermal Science, 2010.

[43] R. Diestel, Graph Theory. Springer, 2017.

[44] GraphPlot, "Graphplot.jl." https://github.com/JuliaPlots/GraphPlot.jl. Accessed: 2021-01-23.

[45] IPCC, "Climate change 2014: Mitigation of climate change. contribution of working group iii to the fifth assessment report of the intergovernmental panel on climate change," 2014.

[46] M. Huber, D. Dimkova, and T. Hamacher, "Integration of wind and solar power in europe: Assessment of flexibility requirements," Energy, vol. 69, pp. 236–246, 2014.

[47] Agora, "Documentation: Prices." https://www.agora-energiewende.de/fileadmin2/Projekte/2019/Jahresauswertung_2019/A-EW_German-Power-Market-2019_Summary_EN.pdf. Accessed: 2021-01-23.

[48] MOSEK, "Modeling cookbook." https://docs.mosek.com/modeling-cookbook/index.html. Accessed: 2021-01-24.

[49] ENS, "Transmissions line data." Data given from the chair ENS @ Technische Universität München (also used in this https://mediatum.ub.tum.de/doc/1544410/337152716379.pdf). Accessed: 2021-01-23.

[50] C. Coffrin, R. Bent, K. Sundar, Y. Ng, and M. Lubin, "Powermodels.jl: An open-source framework for exploring power flow formulations," 2018.