Planner Documentation

version 1.0

July 25, 2022

Contents

Chapter 1

Introduction

The objective of this document is to provide a comprehensive user and development guide of an integrated system enviroment of energy investment planning problems. The integrated system environment, referred to as "Planner", is written in Julia and JuMP, and it is composed by 3 different subsystems:

1. Data manager and model linker
2. Optimization models
3. Solution methodology

The objective of the Planner is to be extremely flexible and user friendly, helping the user to solve the energy planning optimization problems with decomposition methodology (with a particular focus in the novel adaptive Benders with inexact oracles [1]).

Each subsystem will be explained in detail, and then practical examples for different type of users and developers will be presented.
Depending on user needs the recommended read of this document varies.

1.1 First time with the Planner

If you are new with the "Planner" then the recommended process is the following:

1. Read Section 2.1 to understand the path where data, models and solution methodology are contained.
2. Read Section 7.3 to see how to execute the code.
3. Read Chapter 6 to see the scripts with explicit comments on what each line does.
4. Read Section 7.2 to see the models behind the example explained in the scripts.
5. Read Section 2.1.2 and open the algorithm data file (Nicolo/dataXLSX/algorithm.xlsx) to play with different algorithm parameters.
6. Run the model and retrieve results according to Section 7.4.

1.2 Playing with parameters

After successfully executed the "First time with the Planner", the recommendation is to start modifying (only) the parameters for the model, they can include generator characteristics and cost coefficients.

1. Open the data files of Nicolo's example: Nicolo/dataXLSX/investment.xlsx and -operational.xlsx
2. Modify values as required.

3. Execute the main.jl script (Sec. 7.3).

Then try adding new generators by creating additional elements in the operational and investment data. If these generators are contained in a set already defined, and no modifications to the models are neededo only the linking file needs to be updated then:

- Change the Nicolo/dataXLSX/link.xlsx to map the new names between investment and operation.

1.3 Changing the stochastic tree

The case used as a basis (Nicolo) has 3 example stochastic trees. In investment.xlsx, the sheet *structure* defines the tree, and simply by telling the data reader to read e.g. *structure91* will point to a different tree.

The user can modify parameters to be passed in different nodes of the tree, and can also modify probabilities or in general, it can change the structure of the tree. If the structure is changed, the user needs to make sure that the data is correctly defined for the new tree.

A couple of auxiliary files/scripts helps the user to quickly create trees. The general version of it is written in Julia by Nagisa Sughishita and all the documentation and instructions are contained within the script (functions/tree.jl); please refer to it for assistance on tree generation. The output of the general script generator can be generated by running the script treePopulator.jl

The second alternative is to open a XLSX-based tree generator (functions/tree_generator_v2.xlsx) that helps to create regular trees. There is an output sheet that as should be copied to the *structure* sheet in the investment.xlsx

If a modification of the tree is required, then it is enough to modify the data in the corresponding sheet. If a new tree is required it can be directly (explicitly) written in the structure sheet, or it can be assisted by the tree generator scripts, the process is the following:

1. Read the documentation in function/tree.jl
2. Generate XLSX tables for each uncertainty dimension
3. Run treeGenerator.jl to read XLSX tables and output them in a file
4. Copy the output to the structure sheet

1.4 Developing a new optimization model

The recommended process to develop a new optimization model relies first, in being familiar with the planner, and being able to modify parameters, adding new elements and understanding how data is connected.

In addition to those steps, a convenient way would be analysing and comparing 2 different examples included e.g. Nicolo and Hongyu to each other, so the user can learn what is needed when developing new optimization models.

A suggested process to do it is:

1. Read Chapter 3 to understand how optimisation models are created and populated
2. Open and play with 2 different examples: Nicolo and Hongyu
3. Create or modify the data (Chapter 2) that it is required for the model
4. Modify the linkage between models (Chapter 4)

1.5 Developing the solution algorithm

To develop the solution methodology the user needs to understand how the data and models are held, and how to query information from the models.

1. Read the paper [1]
2. Read Section 7.7
3. Read the functions: *solve_Benders!*, *step!* ,*step_0!*, *step_a!*, *step_b!*, *step_c!*, *step_e!*, *step_f!*, contained in functions/load_functions.jl

Chapter 2

Data manager

2.1 Path structure

As explained earlier, there are different modules that constitute the planner; the modules are first called from an julia script. It is easier to explain the structure using a concrete example. The example chosen is called "Nicolo" and corresponds to the examples for the paper [1].
The general structure of the files is shown in Figure 2.1.

Different models can have different elements (e.g. lines, generators, demands, buses, generator types, etcetera) and those elements can have different properties (e.g. ramp rate, capacity, reactance, etcetera). There are two different input format options: XLSX based spreadsheets and CSV single table files. For simplicity, the example "Nicolo" includes data in XLSX spreadsheets. If more information about the CSV files is needed, please refer to the Appendix.

2.1.1 main.xlsx

As Figure 6.1 on page 38 shows, when specifying data, there is a main file (in that example is Nicolo/-dataXLSX/main.xlsx), which contains a list of all the files that will be read and dumped into a dictionary of tables. Figure 6.1 shows the contents of the file in a sheet called "main" which points to other XLSX files. The header row (Row 1) includes 2 fields, Data and Filename. *Data* refers to the name of the dictionary key name where the data is going to be held (In a later section, this will be shown); *Filename* gives the filename where the files are stored. It is important to mention that this list of files is completely customisable and can have different elements to fit the requirements.

Figure 2.2 shows the tables to be read and imported into some data structure.

2.1.2 algorithm.xlsx

algorithm.xlsx contains only 1 problem information sheet (the others are required for the reader as part of the structure) called *general*, in there there are options to tune the solution algorithm. The most important one is "Algorithm" as it defines the solution methodology. Algorithm = 0 chooses to solve the undecomposed version of the planning problem, while options 1 and 2 refer to standard Benders and adaptive Benders, respectively. For more detailed information, in the file there is a readme sheet explaining most of the parameters.

2.1.3 investment.xlsx, operation.xlsx and other general files

investment.xlsx and operation.xlsx are proper problem dependent spreadsheets. They containt elements and properties that can be used when building the model. As it will be explain they have *index*, *parameters* and content sheets; additionally these files include a *sets* sheet which will help the user to build sets and to index the properties that will be required later when building the model.

In these files there can be as many general sheets as needed, which will depend on the specific instance or model to be solved.

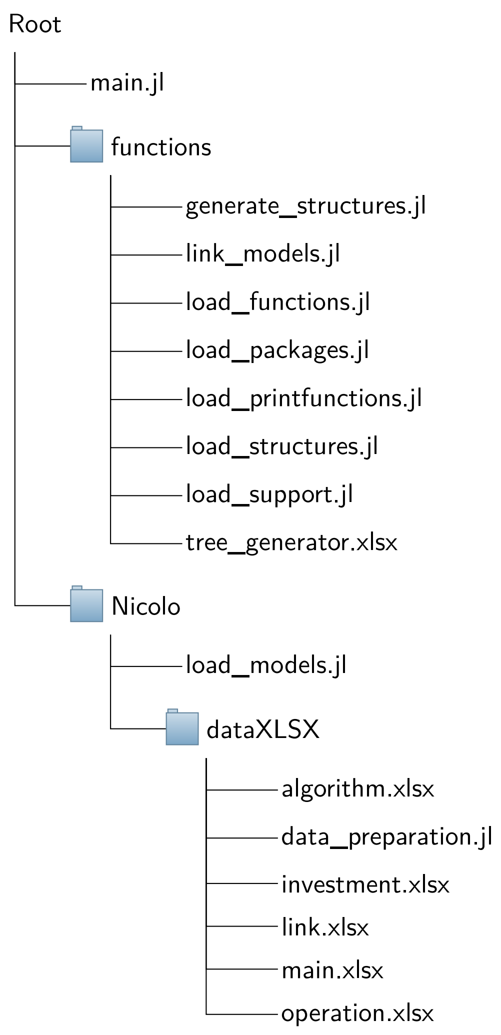
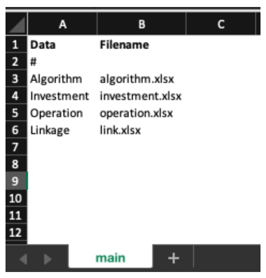
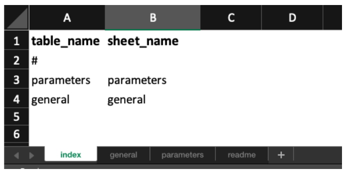


Figure 2.1: Structure of the code and data



	A	B	C
1	Data	Filename	
2	#		
3	Algorithm	algorithm.xlsx	
4	Investment	investment.xlsx	
5	Operation	operation.xlsx	
6	Linkage	link.xlsx	
7			
8			
9			
10			
11			
12			

Figure 2.2: main.xlsx showing the files to read



	A	B	C	D
1	**table_name**	**sheet_name**		
2	#			
3	parameters	parameters		
4	general	general		
5				
6				

Figure 2.3: algorithm.xlsx showing different sheet types

2.1.4 link.xlsx

link.xlsx is a special XLSX spreadsheet and its objective is to link the subproblem and the master problem so they can be solved by decomposition.

As many other files, this file contains an *index* sheet, but also includes unique sheets that will be described later.

2.2 Spreadsheet/CSV Structure

For the data to be read correctly, the XLSX files have a specific structure. In most cases, they will have an *index* sheet, a *parameters* sheet and general data sheets holding other problem specific data. Two special cases are the main.xlsx file and the link.xlsx file. The main.xlsx was explained in Section 2.1.1 and the link.xlsx will be explained in Chapter 4.

Figure 2.3 show an example of a file containing different sheets and sheetypes.

2.2.1 General structure

When read by the generateTablesfromXLSXNew and generateTables (for XLSX files and CSV files, respectively) functions, the tables are imported first as DataFrames and then converted into dictionaries. The use of the external CSV.jl and Dataframes.jl packages, assume that:

- A single table is contained in a sheet.
- The table includes a header row (Row 1).
- The table can have missing content.
- The symbol # is used for lines with comments.

This is true for all the sheets read and contained in the XLSX files, regardless of type (Index, Parameters, content, etcetera).

2.2.2 Index sheet

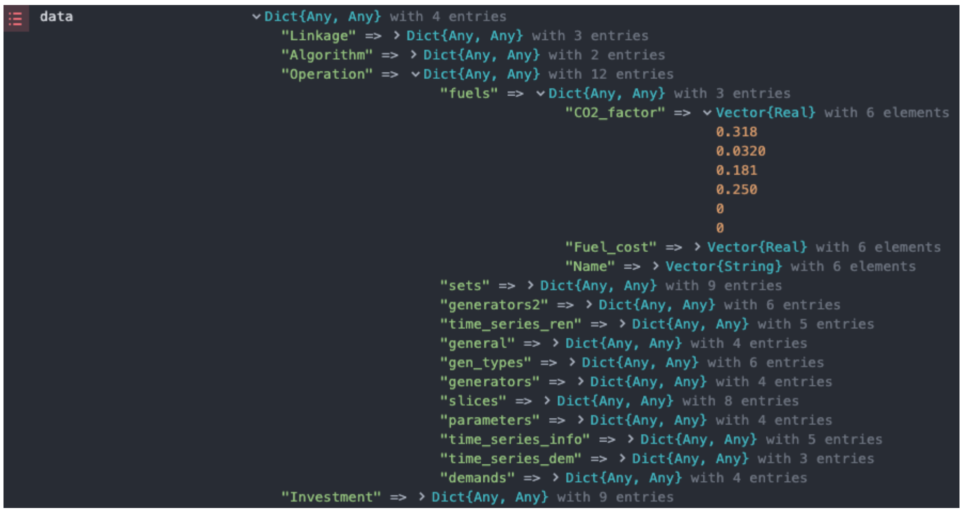
The index sheet informs the data reader with the names of the sheets to be read inside the file and with the dictionary key where the information will be dumped; and with the sheet where data is held.

In Figure 2.3, the active sheet shows a header (marked with bold font), then a comment row denoted by the # symbol at the start of Row 2, and then a list of table names and sheet names. *table_name* refers to the dictionary key in which the data will be held, and *sheet_name* tells which sheet to read for the information.

2.2.3 Parameters sheet

The objective of the sheet is to help the user creating Julia variables that can help the user to access efficiently the information found in the tables. There are custom functions (in particular *loadSets*) for these sheets that will be explained in the "functions" section, and it will be shown an explicit case of its usage later in this document.

The parameters sheet has the columns (headers) *model_name*, *table_name*, *column_name* and *indexing*. *model_name* refers to the name in Julia's structure, *table_name* refers to the dictionary key of where the data is located, *column_name* refers to the column in the table that has the information of the property. If there is no indexing for the property, the *indexing* field will be empty, whereas if the parameters have some indexing (set) it will be specified here.



```
data                     ⌄Dict{Any, Any} with 4 entries
                           "Linkage" => › Dict{Any, Any} with 3 entries
                           "Algorithm" => › Dict{Any, Any} with 2 entries
                           "Operation" => ⌄Dict{Any, Any} with 12 entries
                                           "fuels" => ⌄Dict{Any, Any} with 3 entries
                                                      "CO2_factor" => ⌄Vector{Real} with 6 elements
                                                                       0.318
                                                                       0.0320
                                                                       0.181
                                                                       0.250
                                                                       0
                                                                       0
                                                      "Fuel_cost" => › Vector{Real} with 6 elements
                                                      "Name" => › Vector{String} with 6 elements
                                           "sets" => › Dict{Any, Any} with 9 entries
                                           "generators2" => › Dict{Any, Any} with 6 entries
                                           "time_series_ren" => › Dict{Any, Any} with 5 entries
                                           "general" => › Dict{Any, Any} with 4 entries
                                           "gen_types" => › Dict{Any, Any} with 6 entries
                                           "generators" => › Dict{Any, Any} with 4 entries
                                           "slices" => › Dict{Any, Any} with 8 entries
                                           "parameters" => › Dict{Any, Any} with 4 entries
                                           "time_series_info" => › Dict{Any, Any} with 5 entries
                                           "time_series_dem" => › Dict{Any, Any} with 3 entries
                                           "demands" => › Dict{Any, Any} with 4 entries
                           "Investment" => › Dict{Any, Any} with 9 entries
```

Figure 2.4: Dictionary generated from data files

2.2.4 Sets sheet

There are files that contain a *sets* sheet, which helps the user to form sets (and sets of sets) that will be used later for model building purposes and also they are important when indexing parameters (or properties) with respect to the elements in those sets.

The sets sheet has several columns that give the user flexibility when building sets, subsets and sets of sets. They are used by the function *loadSets* which will explain in detail its usage.

2.2.5 Content sheets

General content sheets can have as many columns and rows as required by the model. The idea of these sheets is that they hold specific problem data, and as such, one of the columns must be defined to match some set (or subset) elements so they can be accessed when needed.

2.2.6 Time-series sheets

The time-series sheets and the *time_series_info* sheet are general sheets but as with the *parameters* and *sets* sheets, there are specific functions that assume special naming conventions and structures, these will be specified in the functions section.

2.3 Functions

Regarding the data manager, there are several custom functions that have been written with the idea to help the user building datastructures which are convenient for model building and model solving. The functions presented in this section are sorted roughly in the order they appear in the execution scripts. These ordering helps the user to have an overview of how the data is logically transformed in the process.

2.3.1 generateTablesfromXLSXNew

- Location: */functions/load_support.jl*
- Description: reads the XLSX files (specified earlier) and dumps the information in a dictionary of dictionaries.
- Usage: *generateTablesfromXLSXNew(path::String,filename::String)* as shown takes two string arguments, the path for the data folder e.g. "Nicolo/dataXLSX" and the name of the main file e.g. "main.xlsx"

In Figure 2.4 the output dictionary (*data*) is shown and for demonstrative purposes the key *Operation* → *fuels* → *CO2_factor* has been expanded to show the contents of the column *CO2_factor* found in the file *Nicolo/dataXLSX/operation.xlsx*, sheet *fuels*.

2.3.2 generateTables

This function has a similar purpose as generateTablesfromXLSXNew, but it assumes that the data is held in separate CSV files. The outputs are exactly the same but the headers in the inputs are slightly modified. This documentation assumes XLSX is the most common data format. For an example involving CSV files, please refer to *Nicolo/case0* example.

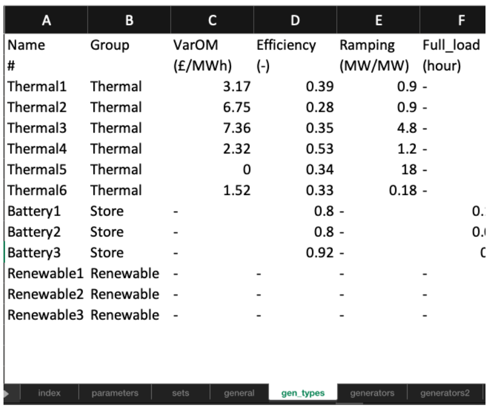
2.3.3 inheritData!

- Location: */functions/load_support.jl*
- Description: transforms the dictionaries by dumping information from one dictionary into another, by matching properties. It has several custom options, such as replacing original data, exporting only a subset of columns, or changing the name of the inherited data.
- Usage: *inheritData!(data::Dict,obj::Pair,matchn::Pair,fields::ArrayString,1;outputChange=[""],overwrite=false)* as shown takes several arguments, these are better explained with a concrete example.

In the example presented here the function is called with:

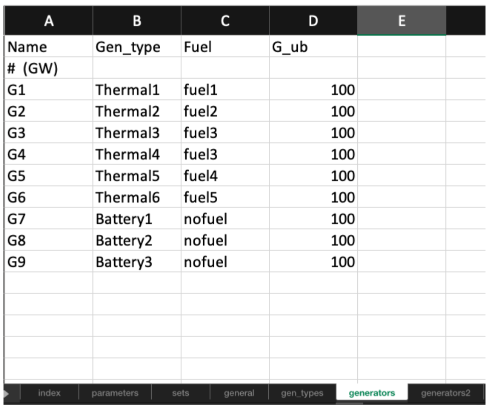
inheritData!(data["Operation"],"gen_types"=>"generators", "Name"=>"Gen_type")

that can be read in the following way: *data["Operation"]* is the dictionary of dictionaries where all the information from operation.xlsx is held, then *"gen_types"=>"generators"* tells that information from the *gen_types* is going to be copied (inherited) to *generators*, by matching the *Name* in *gen_types* field with the field *Gen_type* in *generators*. The example of the transformation of the dictionary is shown in figures 2.5 and 2.6.



A	B	C	D	E	F
Name	Group	VarOM	Efficiency	Ramping	Full_load
#		(£/MWh)	(-)	(MW/MW)	(hour)
Thermal1	Thermal	3.17	0.39	0.9	-
Thermal2	Thermal	6.75	0.28	0.9	-
Thermal3	Thermal	7.36	0.35	4.8	-
Thermal4	Thermal	2.32	0.53	1.2	-
Thermal5	Thermal	0	0.34	18	-
Thermal6	Thermal	1.52	0.33	0.18	-
Battery1	Store	-	0.8	-	0.
Battery2	Store	-	0.8	-	0.
Battery3	Store	-	0.92	-	0
Renewable1	Renewable	-	-	-	-
Renewable2	Renewable	-	-	-	-
Renewable3	Renewable	-	-	-	-

(a) Example table: Generator Types



A	B	C	D	E
Name	Gen_type	Fuel	G_ub	
# (GW)				
G1	Thermal1	fuel1	100	
G2	Thermal2	fuel2	100	
G3	Thermal3	fuel3	100	
G4	Thermal4	fuel3	100	
G5	Thermal5	fuel4	100	
G6	Thermal6	fuel5	100	
G7	Battery1	nofuel	100	
G8	Battery2	nofuel	100	
G9	Battery3	nofuel	100	

(b) Example table: Generators

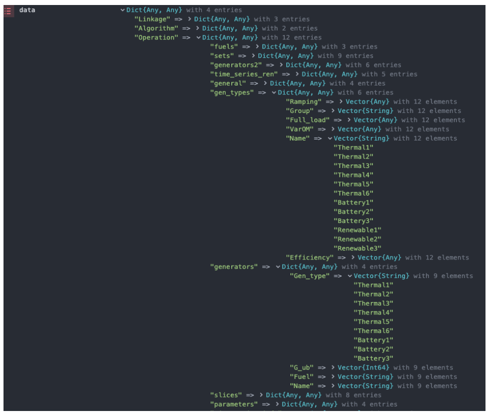
Figure 2.5: Spreadsheet tables

In Figure 2.6a the *data["Operation"]["generators"]* dictionary only contains 4 entries (properties) based on the spreadsheet table shown in Figure 2.5b; once the data has been matched and inherited , the number of properties that present for *generators* has increased to 9 (including fields such as *Ramping* or *Fuel* as seen in Figure 2.6b. Note that in this example, there is little scope to inheriting data, as there is almost the same number of generators as generator types, but in other examples (such as *Hongyu*) there are 60 generators but only 12 generator types, so inheriting data becomes more attractive.

For other version and diverse usage of the *inheritData!* function, please refer to the code in the examples presented later.

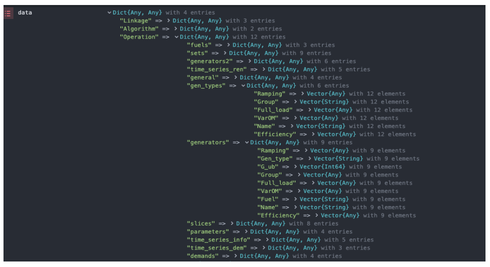
2.3.4 augmentSlices!

- Location: */functions/load_support.jl*
- Description: adds offset and length of the slices for a flatter time-indexed model construction.



```
data                ⌄Dict{Any, Any} with 4 entries
                      "Linkage" => › Dict{Any, Any} with 3 entries
                      "Algorithm" => › Dict{Any, Any} with 2 entries
                      "Operation" => ⌄Dict{Any, Any} with 12 entries
                                       "fuels" => › Dict{Any, Any} with 3 entries
                                       "sets" => › Dict{Any, Any} with 9 entries
                                       "generators2" => › Dict{Any, Any} with 6 entries
                                       "time_series_ren" => › Dict{Any, Any} with 5 entries
                                       "general" => › Dict{Any, Any} with 4 entries
                                       "gen_types" => ⌄Dict{Any, Any} with 6 entries
                                                       "Ramping" => › Vector{Any} with 12 elements
                                                       "Group" => › Vector{String} with 12 elements
                                                       "Full_load" => › Vector{Any} with 12 elements
                                                       "VarOM" => › Vector{Any} with 12 elements
                                                       "Name" => ⌄Vector{String} with 12 elements
                                                                  "Thermal1"
                                                                  "Thermal2"
                                                                  "Thermal3"
                                                                  "Thermal4"
                                                                  "Thermal5"
                                                                  "Thermal6"
                                                                  "Battery1"
                                                                  "Battery2"
                                                                  "Battery3"
                                                                  "Renewable1"
                                                                  "Renewable2"
                                                                  "Renewable3"
                                                       "Efficiency" => › Vector{Any} with 12 elements
                                       "generators" => ⌄Dict{Any, Any} with 4 entries
                                                        "Gen_type" => ⌄Vector{String} with 9 elements
                                                                       "Thermal1"
                                                                       "Thermal2"
                                                                       "Thermal3"
                                                                       "Thermal4"
                                                                       "Thermal5"
                                                                       "Thermal6"
                                                                       "Battery1"
                                                                       "Battery2"
                                                                       "Battery3"
                                                        "G_ub" => › Vector{Int64} with 9 elements
                                                        "Fuel" => › Vector{String} with 9 elements
                                                        "Name" => › Vector{String} with 9 elements
                                       "slices" => › Dict{Any, Any} with 8 entries
                                       "parameters" => › Dict{Any, Any} with 4 entries
```

(a) Data dictionary: before inheritData!



```
data                ⌄Dict{Any, Any} with 4 entries
                      "Linkage" => › Dict{Any, Any} with 3 entries
                      "Algorithm" => › Dict{Any, Any} with 2 entries
                      "Operation" => ⌄Dict{Any, Any} with 12 entries
                                       "fuels" => › Dict{Any, Any} with 3 entries
                                       "sets" => › Dict{Any, Any} with 9 entries
                                       "generators2" => › Dict{Any, Any} with 6 entries
                                       "time_series_ren" => › Dict{Any, Any} with 5 entries
                                       "general" => › Dict{Any, Any} with 4 entries
                                       "gen_types" => ⌄Dict{Any, Any} with 6 entries
                                                       "Ramping" => › Vector{Any} with 12 elements
                                                       "Group" => › Vector{String} with 12 elements
                                                       "Full_load" => › Vector{Any} with 12 elements
                                                       "VarOM" => › Vector{Any} with 12 elements
                                                       "Name" => › Vector{String} with 12 elements
                                                       "Efficiency" => › Vector{Any} with 12 elements
                                       "generators" => ⌄Dict{Any, Any} with 9 entries
                                                        "Ramping" => › Vector{Any} with 9 elements
                                                        "Gen_type" => › Vector{String} with 9 elements
                                                        "G_ub" => › Vector{Int64} with 9 elements
                                                        "Group" => › Vector{Any} with 9 elements
                                                        "Full_load" => › Vector{Any} with 9 elements
                                                        "VarOM" => › Vector{Any} with 9 elements
                                                        "Fuel" => › Vector{String} with 9 elements
                                                        "Name" => › Vector{String} with 9 elements
                                                        "Efficiency" => › Vector{Any} with 9 elements
                                       "slices" => › Dict{Any, Any} with 8 entries
                                       "parameters" => › Dict{Any, Any} with 4 entries
                                       "time_series_info" => › Dict{Any, Any} with 5 entries
                                       "time_series_dem" => › Dict{Any, Any} with 3 entries
                                       "demands" => › Dict{Any, Any} with 4 entries
```

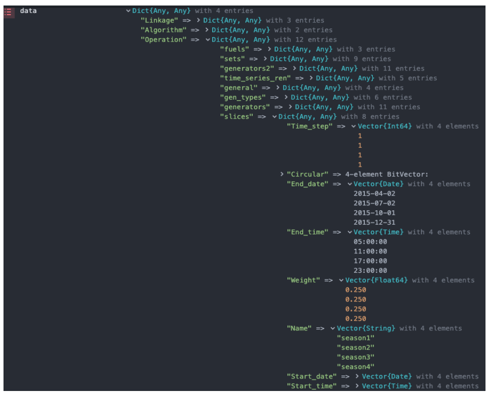
(b) Data dictionary: after inheritData!

Figure 2.6: inheritData! dictionary modification

- Usage: *augmentSlices!(datawpath::Dict,table_sl::String)* as shown takes two arguments, the dictionary (in the level that should contain the slice information as one of the keys), and the key of the dictionary containing the slice information

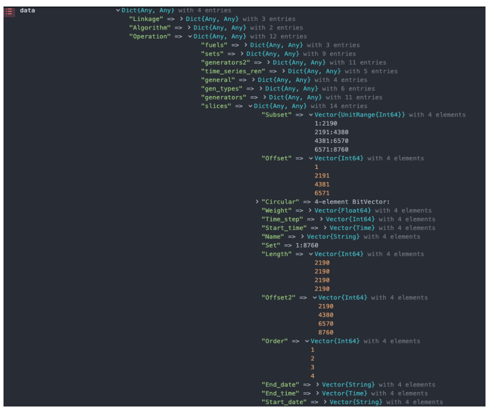
This function is tailored for models that include operational "Slices", which are subsets of periods weighted to represent longer timescales, such as a whole year. While the operational models can be constructed using indexing based on slices and then periods, for these examples the decision was to compact the indexing by calculating the offsets that determine the start and end of each slice.

The effects of calling the function can bee seen in Figure 2.7 corresponding to the data table 2.8. The function call in this example is: *augmentSlices!(data["Operation"],"slices")*



```
data                ⌄Dict{Any, Any} with 4 entries
                      "Linkage" => › Dict{Any, Any} with 3 entries
                      "Algorithm" => › Dict{Any, Any} with 2 entries
                      "Operation" => ⌄Dict{Any, Any} with 12 entries
                                       "fuels" => › Dict{Any, Any} with 3 entries
                                       "sets" => › Dict{Any, Any} with 9 entries
                                       "generators2" => › Dict{Any, Any} with 11 entries
                                       "time_series_ren" => › Dict{Any, Any} with 5 entries
                                       "general" => › Dict{Any, Any} with 4 entries
                                       "gen_types" => › Dict{Any, Any} with 6 entries
                                       "generators" => › Dict{Any, Any} with 11 entries
                                       "slices" => ⌄Dict{Any, Any} with 8 entries
                                                    "Time_step" => ⌄Vector{Int64} with 4 elements
                                                                     1
                                                                     1
                                                                     1
                                                                     1
                                                  › "Circular" => 4-element BitVector:
                                                    "End_date" => ⌄Vector{Date} with 4 elements
                                                                    2015-04-02
                                                                    2015-07-02
                                                                    2015-10-01
                                                                    2015-12-31
                                                    "End_time" => ⌄Vector{Time} with 4 elements
                                                                    05:00:00
                                                                    11:00:00
                                                                    17:00:00
                                                                    23:00:00
                                                    "Weight" => ⌄Vector{Float64} with 4 elements
                                                                  0.250
                                                                  0.250
                                                                  0.250
                                                                  0.250
                                                    "Name" => ⌄Vector{String} with 4 elements
                                                                "season1"
                                                                "season2"
                                                                "season3"
                                                                "season4"
                                                    "Start_date" => › Vector{Date} with 4 elements
                                                    "Start_time" => › Vector{Time} with 4 elements
```

(a) Data dictionary: before augmentSlices!



```
data                ⌄Dict{Any, Any} with 4 entries
                      "Linkage" => › Dict{Any, Any} with 3 entries
                      "Algorithm" => › Dict{Any, Any} with 2 entries
                      "Operation" => ⌄Dict{Any, Any} with 12 entries
                                       "fuels" => › Dict{Any, Any} with 3 entries
                                       "sets" => › Dict{Any, Any} with 9 entries
                                       "generators2" => › Dict{Any, Any} with 11 entries
                                       "time_series_ren" => › Dict{Any, Any} with 5 entries
                                       "general" => › Dict{Any, Any} with 4 entries
                                       "gen_types" => › Dict{Any, Any} with 6 entries
                                       "generators" => › Dict{Any, Any} with 11 entries
                                       "slices" => ⌄Dict{Any, Any} with 14 entries
                                                    "Subset" => ⌄Vector{UnitRange{Int64}} with 4 elements
                                                                  1:2190
                                                                  2191:4380
                                                                  4381:6570
                                                                  6571:8760
                                                    "Offset" => ⌄Vector{Int64} with 4 elements
                                                                  1
                                                                  2191
                                                                  4381
                                                                  6571
                                                  › "Circular" => 4-element BitVector:
                                                    "Weight" => › Vector{Float64} with 4 elements
                                                    "Time_step" => › Vector{Int64} with 4 elements
                                                    "Start_time" => › Vector{Time} with 4 elements
                                                    "Name" => › Vector{String} with 4 elements
                                                    "Set" => 1:8760
                                                    "Length" => ⌄Vector{Int64} with 4 elements
                                                                  2190
                                                                  2190
                                                                  2190
                                                                  2190
                                                    "Offset2" => ⌄Vector{Int64} with 4 elements
                                                                   2190
                                                                   4380
                                                                   6570
                                                                   8760
                                                    "Order" => ⌄Vector{Int64} with 4 elements
                                                                 1
                                                                 2
                                                                 3
                                                                 4
                                                    "End_date" => › Vector{String} with 4 elements
                                                    "End_time" => › Vector{Time} with 4 elements
                                                    "Start_date" => › Vector{String} with 4 elements
```

(b) Data dictionary: after augmentSlices!

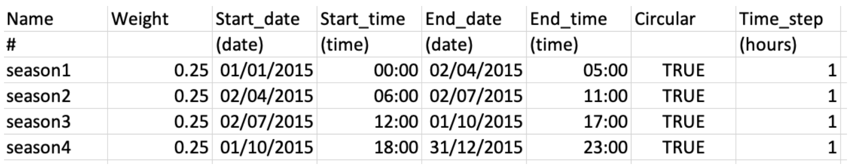
Figure 2.7: augmentSlices! dictionary modification

The newly added elements to the dictionary will include, the Offset of each slice, the set with the total number of periods and the subsets ranges integer indexed.

This function has some important implied assumptions. First, the slices need to include a *Start_time*, *Start_date*, *End_time*, *End_date* and *Time_step* columns, which if using the XLSX spreadsheets, it need to have the cell types in Excel, so the data is read correctly. Then in each slice, there is a single definition of time-step, meaning that the offsets can be calculated with the difference of end-time, end-date minus start-time, start-date divided by the time-step.

2.3.5 genTimeSeries!

- Location: */functions/load_support.jl*
- Description: generates the time-series that match the augmented Slice data (output of the *augmentedSlices!* function)



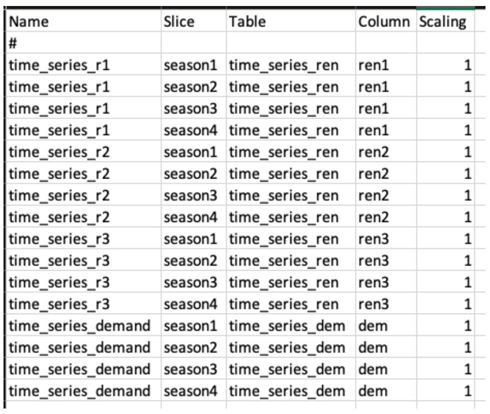
Name	Weight	Start_date	Start_time	End_date	End_time	Circular	Time_step
#		(date)	(time)	(date)	(time)		(hours)
season1	0.25	01/01/2015	00:00	02/04/2015	05:00	TRUE	1
season2	0.25	02/04/2015	06:00	02/07/2015	11:00	TRUE	1
season3	0.25	02/07/2015	12:00	01/10/2015	17:00	TRUE	1
season4	0.25	01/10/2015	18:00	31/12/2015	23:00	TRUE	1

Figure 2.8: Example table: Slices

- Usage: *genTimeSeries!(datawpath::Dict,table_sl::String,info_table::String,output_dict::String)* as shown takes 4 arguments, the dictionary (in the level that should contain the slice information as one of the keys), the key of the dictionary containing the slice information, the key of the timeseries info, and finally the wanted output key. There are additional options for formatting, that can be checked directly in the function definition

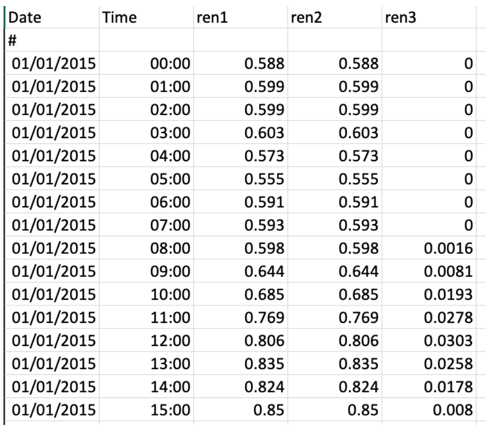
The function will modify the table as shown in Figures 2.9 and 2.10. In this example the function called is:

genTimeSeries!(data["Operation"], "slices", "time_series_info", "time_series")



Name	Slice	Table	Column	Scaling
#				
time_series_r1	season1	time_series_ren	ren1	1
time_series_r1	season2	time_series_ren	ren1	1
time_series_r1	season3	time_series_ren	ren1	1
time_series_r1	season4	time_series_ren	ren1	1
time_series_r2	season1	time_series_ren	ren2	1
time_series_r2	season2	time_series_ren	ren2	1
time_series_r2	season3	time_series_ren	ren2	1
time_series_r2	season4	time_series_ren	ren2	1
time_series_r3	season1	time_series_ren	ren3	1
time_series_r3	season2	time_series_ren	ren3	1
time_series_r3	season3	time_series_ren	ren3	1
time_series_r3	season4	time_series_ren	ren3	1
time_series_demand	season1	time_series_dem	dem	1
time_series_demand	season2	time_series_dem	dem	1
time_series_demand	season3	time_series_dem	dem	1
time_series_demand	season4	time_series_dem	dem	1

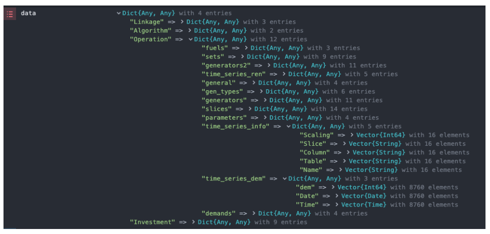
(a) Example table: Time series info



Date	Time	ren1	ren2	ren3
#				
01/01/2015	00:00	0.588	0.588	0
01/01/2015	01:00	0.599	0.599	0
01/01/2015	02:00	0.599	0.599	0
01/01/2015	03:00	0.603	0.603	0
01/01/2015	04:00	0.573	0.573	0
01/01/2015	05:00	0.555	0.555	0
01/01/2015	06:00	0.591	0.591	0
01/01/2015	07:00	0.593	0.593	0
01/01/2015	08:00	0.598	0.598	0.0016
01/01/2015	09:00	0.644	0.644	0.0081
01/01/2015	10:00	0.685	0.685	0.0193
01/01/2015	11:00	0.769	0.769	0.0278
01/01/2015	12:00	0.806	0.806	0.0303
01/01/2015	13:00	0.835	0.835	0.0258
01/01/2015	14:00	0.824	0.824	0.0178
01/01/2015	15:00	0.85	0.85	0.008

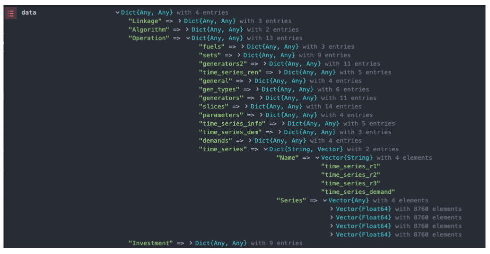
(b) Example table: Time series proper data

Figure 2.9: Time series spreadsheet tables



```
data          ⌄Dict{Any, Any} with 4 entries
                "Linkage" => › Dict{Any, Any} with 3 entries
                "Algorithm" => › Dict{Any, Any} with 2 entries
                "Operation" => ⌄Dict{Any, Any} with 12 entries
                           "fuels" => › Dict{Any, Any} with 3 entries
                           "sets" => › Dict{Any, Any} with 9 entries
                           "generators2" => › Dict{Any, Any} with 11 entries
                           "time_series_ren" => › Dict{Any, Any} with 5 entries
                           "general" => › Dict{Any, Any} with 4 entries
                           "gen_types" => › Dict{Any, Any} with 6 entries
                           "generators" => › Dict{Any, Any} with 11 entries
                           "slices" => › Dict{Any, Any} with 14 entries
                           "parameters" => › Dict{Any, Any} with 4 entries
                           "time_series_info" => ⌄Dict{Any, Any} with 5 entries
                                          "Scaling" => › Vector{Int64} with 16 elements
                                          "Slice" => › Vector{String} with 16 elements
                                          "Column" => › Vector{String} with 16 elements
                                          "Table" => › Vector{String} with 16 elements
                                          "Name" => › Vector{String} with 16 elements
                           "time_series_dem" => ⌄Dict{Any, Any} with 3 entries
                                         "dem" => › Vector{Int64} with 8760 elements
                                         "Date" => › Vector{Date} with 8760 elements
                                         "Time" => › Vector{Time} with 8760 elements
                           "demands" => › Dict{Any, Any} with 4 entries
                "Investment" => › Dict{Any, Any} with 9 entries
```

(a) Data dictionary: before genTimeSeries!



```
data          ⌄Dict{Any, Any} with 4 entries
                "Linkage" => › Dict{Any, Any} with 3 entries
                "Algorithm" => › Dict{Any, Any} with 2 entries
                "Operation" => ⌄Dict{Any, Any} with 13 entries
                           "fuels" => › Dict{Any, Any} with 3 entries
                           "sets" => › Dict{Any, Any} with 9 entries
                           "generators2" => › Dict{Any, Any} with 11 entries
                           "time_series_ren" => › Dict{Any, Any} with 5 entries
                           "general" => › Dict{Any, Any} with 4 entries
                           "gen_types" => › Dict{Any, Any} with 6 entries
                           "generators" => › Dict{Any, Any} with 11 entries
                           "slices" => › Dict{Any, Any} with 14 entries
                           "parameters" => › Dict{Any, Any} with 4 entries
                           "time_series_info" => › Dict{Any, Any} with 5 entries
                           "time_series_dem" => › Dict{Any, Any} with 3 entries
                           "demands" => › Dict{Any, Any} with 4 entries
                           "time_series" => ⌄Dict{String, Vector} with 2 entries
                                     "Name" => ⌄Vector{String} with 4 elements
                                           "time_series_r1"
                                           "time_series_r2"
                                           "time_series_r3"
                                           "time_series_demand"
                                     "Series" => ⌄Vector{Any} with 4 elements
                                            › Vector{Float64} with 8760 elements
                                            › Vector{Float64} with 8760 elements
                                            › Vector{Float64} with 8760 elements
                                            › Vector{Float64} with 8760 elements
                "Investment" => › Dict{Any, Any} with 9 entries
```

(b) Data dictionary: after genTimeSeries!

Figure 2.10: genTimeSeries! dictionary modification

In the output dictionary (Figure 2.10b) a new key has been created with the name of *time_series* which holds the sorted information for the time series.

2.3.6 scaleData!

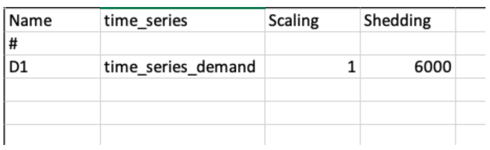
- Location: */functions/load_support.jl*
- Description: multiplies the elements of a time series by a factor

- Usage: *scaleData!(datawpath::Dict,field::String,time_series::String,scalingField::String)* as shown takes 4 arguments, the dictionary (in the level that should contain the slice information as one of the keys), the key of the dictionary containing the information to be scaled information, the key of the time series data, and finally the scaling key where the factors are held.

The idea of the scaling is to be able to independently (from the time series) have the ability to scale the a whole profile. In a problem where 2 demands are similar in behaviour but one applies to a larger area and may have a larger scaling, a function like this is useful.

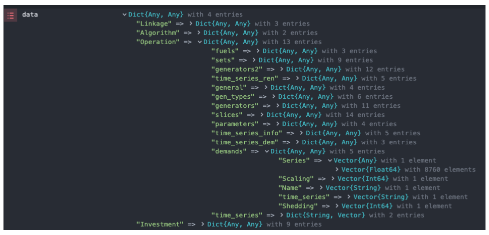
In Figure 2.11, the structure of the example for data and dictionary of demand can be seen, the figures do not show the way the data has been transformed but show the structure. In this example, the function called is:

scaleData!(data["Operation"],"demands","Series","Scaling")



Name	time_series	Scaling	Shedding
#			
D1	time_series_demand	1	6000

(a) Data table: Demands



```
data        ⌄Dict{Any, Any} with 4 entries
              "Linkage" => › Dict{Any, Any} with 3 entries
              "Algorithm" => › Dict{Any, Any} with 2 entries
              "Operation" => ⌄Dict{Any, Any} with 13 entries
                         "fuels" => › Dict{Any, Any} with 3 entries
                         "sets" => › Dict{Any, Any} with 9 entries
                         "generators2" => › Dict{Any, Any} with 12 entries
                         "time_series_ren" => › Dict{Any, Any} with 5 entries
                         "general" => › Dict{Any, Any} with 4 entries
                         "gen_types" => › Dict{Any, Any} with 6 entries
                         "generators" => › Dict{Any, Any} with 11 entries
                         "slices" => › Dict{Any, Any} with 14 entries
                         "parameters" => › Dict{Any, Any} with 4 entries
                         "time_series_info" => › Dict{Any, Any} with 5 entries
                         "time_series_dem" => › Dict{Any, Any} with 3 entries
                         "demands" => ⌄Dict{Any, Any} with 5 entries
                                  "Series" => ⌄Vector{Any} with 1 element
                                                › Vector{Float64} with 8760 elements
                                  "Scaling" => › Vector{Int64} with 1 element
                                  "Name" => › Vector{String} with 1 element
                                  "time_series" => › Vector{String} with 1 element
                                  "Shedding" => › Vector{Int64} with 1 element
                         "time_series" => › Dict{String, Vector} with 2 entries
              "Investment" => › Dict{Any, Any} with 9 entries
```

(b) Data dictionary: Demands

Figure 2.11: scaleData! dictionary area

2.3.7 combineTables

- Location: */functions/load_support.jl*
- Description: joins data of different dictionary (tables) into one.
- Usage: *combineTables(datawpath::Dict,outputKey::String,[inputkeys]::VectorString,uniqueField::String;options* as shown takes 4 arguments plus different options: the dictionary (in the level that should contain the table information as one of the keys), the output key of the dictionary, the list of tables to be joined, and finally the field that is considered as unique.

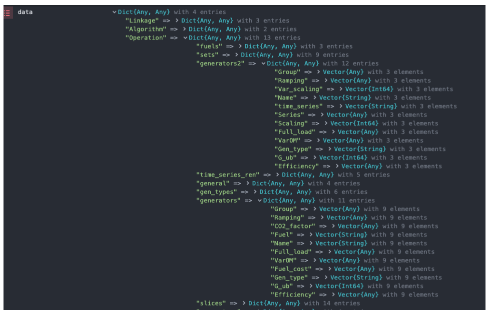
On the left-hand side of Figure 2.12 the dictionary shows 2 entries coming from different spreadsheets *generators* and *generators2*. According to the description of the function, as it is called:

combineTables(data["Operation"],"generators",["generators","generators2"],"Name";deleteSource=true)

The information found in *data["Operation"]* will have an output key of *"generators"* by combining the information found in *"generators"* and *"generators2"* where the unique field is the *"Name"* entry and after that it deletes the original tables. As seen in the figure, the original length of the entries in generators and generators2 where 9 and 3, respectively; and the output table has 12 elements (9+3). Other important takeaways are: First, the unique field was name, in this example there are no repeated names, in case there are repeated names, the information of the later table (in the order of the [inputKeys] vector) will overwrite the information of the first table; the second thing to notice is the additional of fields; if the tables have different headers, the output table will have all the headers found in the 2 tables (with missing values, if missing). In this example the field *Var_scaling* exists in *generators* table and then in the output table. *deleteSource=true* deletes the original table (although in this example the output table has the same name as one of the components).

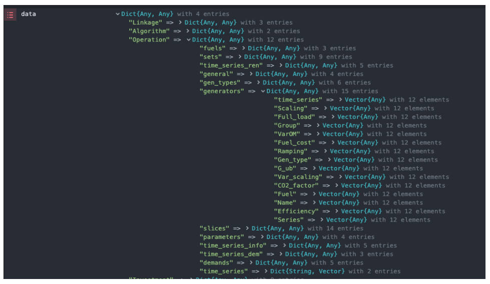
2.3.8 loadSets

- Location: */functions/load_support.jl*
- Description: returns a dictionary with the sets specified as in the *sets* spreadsheet.



```
data                 ⌄Dict{Any, Any} with 4 entries
                       "Linkage" => › Dict{Any, Any} with 3 entries
                       "Algorithm" => › Dict{Any, Any} with 2 entries
                       "Operation" => ⌄Dict{Any, Any} with 13 entries
                                        "fuels" => › Dict{Any, Any} with 3 entries
                                        "sets" => › Dict{Any, Any} with 9 entries
                                        "generators2" => ⌄Dict{Any, Any} with 12 entries
                                                          "Group" => › Vector{Any} with 3 elements
                                                          "Ramping" => › Vector{Any} with 3 elements
                                                          "Var_scaling" => › Vector{Int64} with 3 elements
                                                          "Name" => › Vector{String} with 3 elements
                                                          "time_series" => › Vector{String} with 3 elements
                                                          "Series" => › Vector{Any} with 3 elements
                                                          "Scaling" => › Vector{Int64} with 3 elements
                                                          "Full_load" => › Vector{Any} with 3 elements
                                                          "VarOM" => › Vector{Any} with 3 elements
                                                          "Gen_type" => › Vector{String} with 3 elements
                                                          "G_ub" => › Vector{Int64} with 3 elements
                                                          "Efficiency" => › Vector{Any} with 3 elements
                                        "time_series_ren" => › Dict{Any, Any} with 5 entries
                                        "general" => › Dict{Any, Any} with 4 entries
                                        "gen_types" => › Dict{Any, Any} with 6 entries
                                        "generators" => ⌄Dict{Any, Any} with 11 entries
                                                         "Group" => › Vector{Any} with 9 elements
                                                         "Ramping" => › Vector{Any} with 9 elements
                                                         "CO2_factor" => › Vector{Any} with 9 elements
                                                         "Fuel" => › Vector{String} with 9 elements
                                                         "Name" => › Vector{String} with 9 elements
                                                         "Full_load" => › Vector{Any} with 9 elements
                                                         "VarOM" => › Vector{Any} with 9 elements
                                                         "Fuel_cost" => › Vector{Any} with 9 elements
                                                         "Gen_type" => › Vector{String} with 9 elements
                                                         "G_ub" => › Vector{Int64} with 9 elements
                                                         "Efficiency" => › Vector{Any} with 9 elements
                                        "slices" => › Dict{Any, Any} with 14 entries
```

(a) 2 entries: generators and generators2



```
data                 ⌄Dict{Any, Any} with 4 entries
                       "Linkage" => › Dict{Any, Any} with 3 entries
                       "Algorithm" => › Dict{Any, Any} with 2 entries
                       "Operation" => ⌄Dict{Any, Any} with 12 entries
                                        "fuels" => › Dict{Any, Any} with 3 entries
                                        "sets" => › Dict{Any, Any} with 9 entries
                                        "time_series_ren" => › Dict{Any, Any} with 5 entries
                                        "general" => › Dict{Any, Any} with 4 entries
                                        "gen_types" => › Dict{Any, Any} with 6 entries
                                        "generators" => ⌄Dict{Any, Any} with 15 entries
                                                         "time_series" => › Vector{Any} with 12 elements
                                                         "Scaling" => › Vector{Any} with 12 elements
                                                         "Full_load" => › Vector{Any} with 12 elements
                                                         "Group" => › Vector{Any} with 12 elements
                                                         "VarOM" => › Vector{Any} with 12 elements
                                                         "Fuel_cost" => › Vector{Any} with 12 elements
                                                         "Ramping" => › Vector{Any} with 12 elements
                                                         "Gen_type" => › Vector{Any} with 12 elements
                                                         "G_ub" => › Vector{Any} with 12 elements
                                                         "Var_scaling" => › Vector{Any} with 12 elements
                                                         "CO2_factor" => › Vector{Any} with 12 elements
                                                         "Fuel" => › Vector{Any} with 12 elements
                                                         "Name" => › Vector{Any} with 12 elements
                                                         "Efficiency" => › Vector{Any} with 12 elements
                                                         "Series" => › Vector{Any} with 12 elements
                                        "slices" => › Dict{Any, Any} with 14 entries
                                        "parameters" => › Dict{Any, Any} with 4 entries
                                        "time_series_info" => › Dict{Any, Any} with 5 entries
                                        "time_series_dem" => › Dict{Any, Any} with 3 entries
                                        "demands" => › Dict{Any, Any} with 5 entries
                                        "time_series" => › Dict{String, Vector} with 2 entries
```

(b) 1 entry (combined): generators

Figure 2.12: combineTables dictionary modifications

- Usage: *loadSets(datawpath::Dict,setSheet::String)::Dict* as shown takes only 2 arguments (dictionary at the level and the name of the table to read the sets from; and returns a dictionary).

This function assumes a fix structure of the sets table (dictionary and datasheet). The needed columns are explained here.

1. model_name: name of the set that will be created in Julia's memory.
2. element_table: table of elements which contains the list of elements where the set is going to be formed.
3. indexing_table: table of elements which contains the indexing of elements (for sets of elements, it is the same as the element_table, when forming sets of sets, this is different as it will be shown later in this subsection).
4. indexing_property: name of the property of the indexing, e.g. name of generators.
5. filtergroup_#: it helps to filter out elements based on some matched property, the # means that several filters can be added (they are and conditions), a example using two filters is presented later.
6. filterval_#: the filtering condition of the filtergroup_#.
7. matching_property: used for sets of sets, by matching the property to be indexed (later shown with an example).
8. inner_indexing: used for sets of sets, to indicate what is the field for the inner indexing (it will be clearer with the example).

First, the simple case involving only sets and subsets is presented for the table coming from the spreadsheet shown in Figure 2.13a, and the output is shown in Figure 2.13b. The function call is:

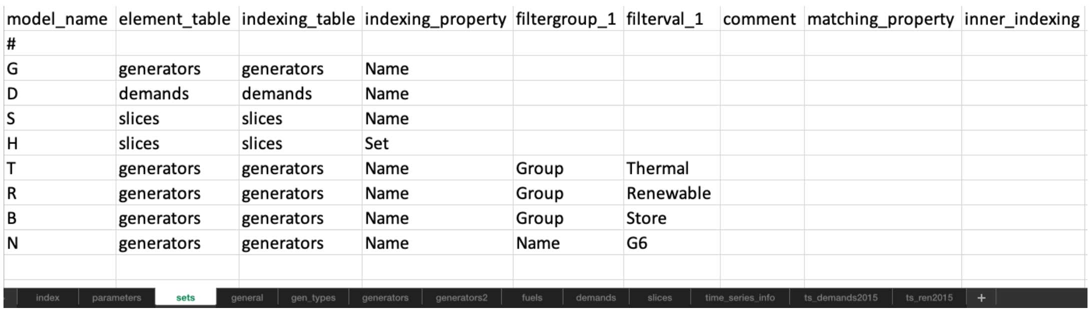
psDict=loadSets(data[”Operation”],”sets”)

In the Figure (2.13), the complete set of generators indexed by name is in the key *”G”*, and then a subset of Thermal generators is stored in the key *”T”*, by filtering all the generators that have the *”Group==Thermal”*. Another example is the nuclear subset, that only includes a single generator that matches the name of *”G6”*.

The second “more” complicated example comes from a different example instance (Hongyu/dataXLSX), where there are several areas (or regions), and then for building constraints, such as the power balance that needs to be done by area, it is convenient to form sets of sets. Figure 2.14 shows the data and output.

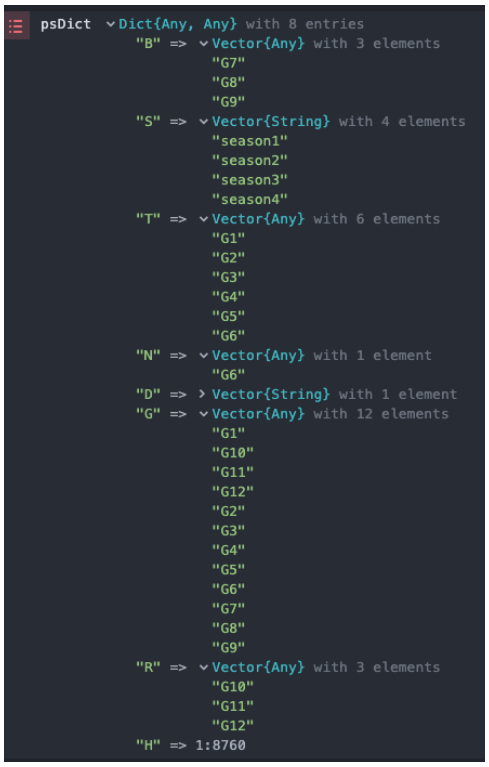
This example is interesting. In one hand, the are more filtering properties, for example, the set *”HS”* gathers the subset of generators, which are of type storage and then of subtype heat, showing than more than 1 filter can be used.

The set *”A”* (expanded in Figure 2.14b) includes 5 areas, and then the key *”T_A”* is indexed first by area and then it contains the elements in that area.



model_name	element_table	indexing_table	indexing_property	filtergroup_1	filterval_1	comment	matching_property	inner_indexing
#								
G	generators	generators	Name					
D	demands	demands	Name					
S	slices	slices	Name					
H	slices	slices	Set					
T	generators	generators	Name	Group	Thermal			
R	generators	generators	Name	Group	Renewable			
B	generators	generators	Name	Group	Store			
N	generators	generators	Name	Name	G6			

(a) Data table: Sets



```
psDict  Dict{Any, Any} with 8 entries
          "B"  =>  Vector{Any} with 3 elements
                   "G7"
                   "G8"
                   "G9"
          "S"  =>  Vector{String} with 4 elements
                   "season1"
                   "season2"
                   "season3"
                   "season4"
          "T"  =>  Vector{Any} with 6 elements
                   "G1"
                   "G2"
                   "G3"
                   "G4"
                   "G5"
                   "G6"
          "N"  =>  Vector{Any} with 1 element
                   "G6"
          "D"  =>  Vector{String} with 1 element
          "G"  =>  Vector{Any} with 12 elements
                   "G1"
                   "G10"
                   "G11"
                   "G12"
                   "G2"
                   "G3"
                   "G4"
                   "G5"
                   "G6"
                   "G7"
                   "G8"
                   "G9"
          "R"  =>  Vector{Any} with 3 elements
                   "G10"
                   "G11"
                   "G12"
          "H"  =>  1:8760
```

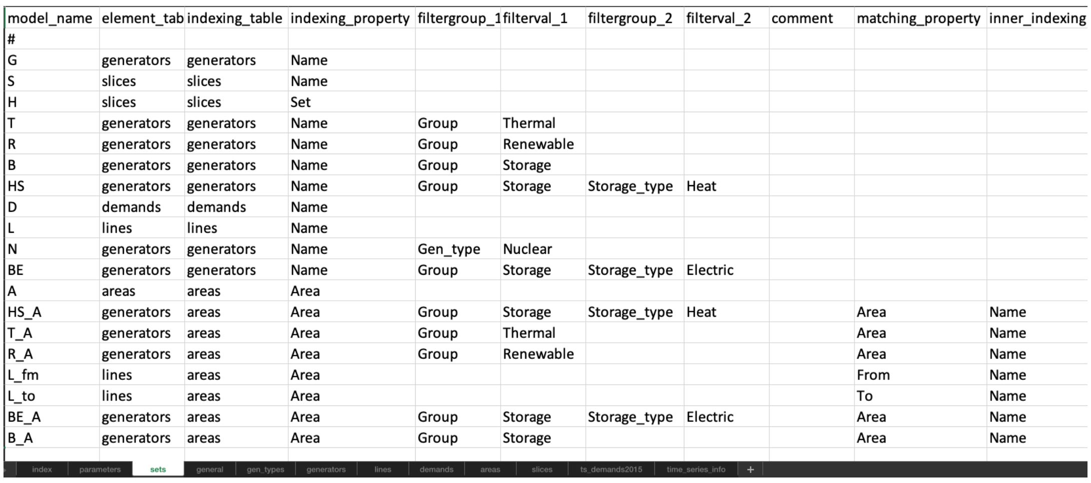
(b) Output dictionary: Sets

Figure 2.13: loadSets function

Analysing the entry on the spreadsheet tells the following: model_name (name in model) - *"T_A"*, then element_table - "generators" refers to the table proper elements in the inner indexing, the indexing_table - "areas" refers to the outer indexing elements, indexing_property - "Area" refers to the outer indexing property, the filter groups function like in the previous example (for the inner indexing), and then the matching property means that the genrators that have been assigned to the "Area" (in the generators table) will then be matched to the outer indexing. Finally, the inner indexing refers to the unique indexing of generators, which in this case is "Name". The description is a bit complicated and maybe anti-natural, the best way to understand it is running the example cases which use different combination of elements of this table.

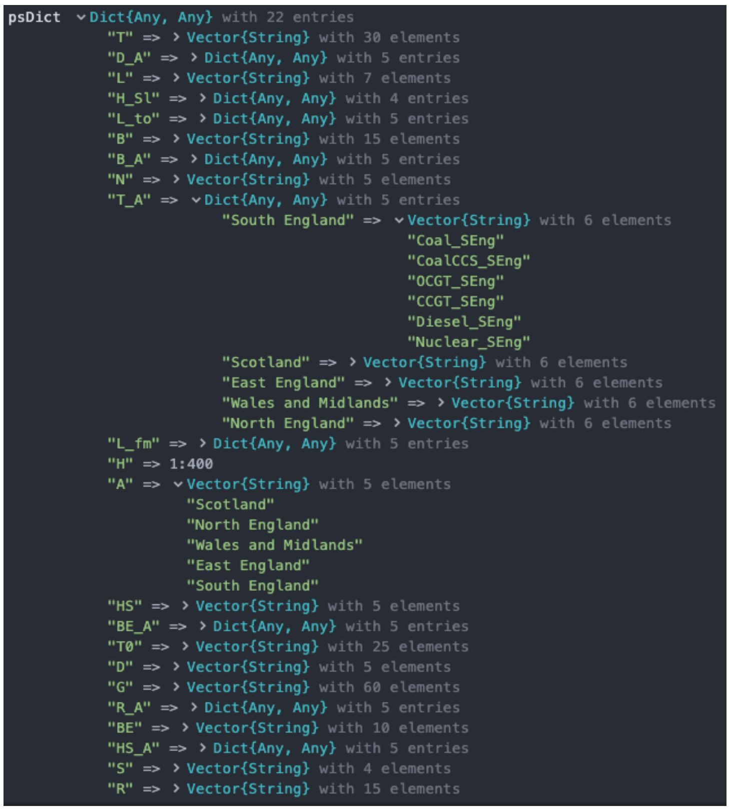
2.3.9 diffSets!

- Location: */functions/load_support.jl*
- Description: removes elements of a set and assigns the remaining elements to a new set (or subset).
- Usage: *diffSets!(dict::Dict, newset::String, sets::Vector{String}* as shown takes 3 arguments, transforming the first one, adding a new key entry with the difference of the sets present in the vector of sets.



model_name	element_tab	indexing_table	indexing_property	filtergroup_1	filterval_1	filtergroup_2	filterval_2	comment	matching_property	inner_indexing
#										
G	generators	generators	Name							
S	slices	slices	Name							
H	slices	slices	Set							
T	generators	generators	Name	Group	Thermal					
R	generators	generators	Name	Group	Renewable					
B	generators	generators	Name	Group	Storage					
HS	generators	generators	Name	Group	Storage	Storage_type	Heat			
D	demands	demands	Name							
L	lines	lines	Name							
N	generators	generators	Name	Gen_type	Nuclear					
BE	generators	generators	Name	Group	Storage	Storage_type	Electric			
A	areas	areas	Area							
HS_A	generators	areas	Area	Group	Storage	Storage_type	Heat		Area	Name
T_A	generators	areas	Area	Group	Thermal				Area	Name
R_A	generators	areas	Area	Group	Renewable				Area	Name
L_fm	lines	areas	Area						From	Name
L_to	lines	areas	Area						To	Name
BE_A	generators	areas	Area	Group	Storage	Storage_type	Electric		Area	Name
B_A	generators	areas	Area	Group	Storage				Area	Name

(a) Data table: Sets (Example 2)



```
psDict  ⌄Dict{Any, Any} with 22 entries
           "T" => › Vector{String} with 30 elements
           "D_A" => › Dict{Any, Any} with 5 entries
           "L" => › Vector{String} with 7 elements
           "H_Sl" => › Dict{Any, Any} with 4 entries
           "L_to" => › Dict{Any, Any} with 5 entries
           "B" => › Vector{String} with 15 elements
           "B_A" => › Dict{Any, Any} with 5 entries
           "N" => › Vector{String} with 5 elements
           "T_A" => ⌄Dict{Any, Any} with 5 entries
                       "South England" => ⌄Vector{String} with 6 elements
                                           "Coal_SEng"
                                           "CoalCCS_SEng"
                                           "OCGT_SEng"
                                           "CCGT_SEng"
                                           "Diesel_SEng"
                                           "Nuclear_SEng"
                       "Scotland" => › Vector{String} with 6 elements
                       "East England" => › Vector{String} with 6 elements
                       "Wales and Midlands" => › Vector{String} with 6 elements
                       "North England" => › Vector{String} with 6 elements
           "L_fm" => › Dict{Any, Any} with 5 entries
           "H" => 1:400
           "A" => ⌄Vector{String} with 5 elements
                    "Scotland"
                    "North England"
                    "Wales and Midlands"
                    "East England"
                    "South England"
           "HS" => › Vector{String} with 5 elements
           "BE_A" => › Dict{Any, Any} with 5 entries
           "T0" => › Vector{String} with 25 elements
           "D" => › Vector{String} with 5 elements
           "G" => › Vector{String} with 60 elements
           "R_A" => › Dict{Any, Any} with 5 entries
           "BE" => › Vector{String} with 10 elements
           "HS_A" => › Dict{Any, Any} with 5 entries
           "S" => › Vector{String} with 4 elements
           "R" => › Vector{String} with 15 elements
```

(b) Output dictionary: Sets and sets of sets (Example 2)

Figure 2.14: loadSets function

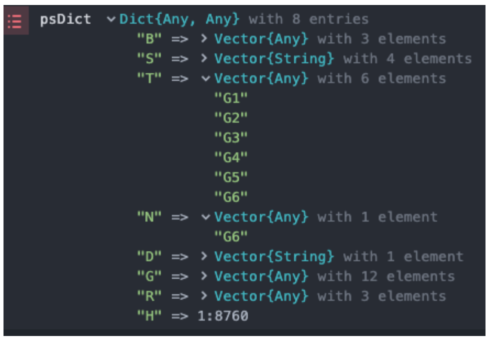
In this example the call is:

diffSets!(psDict,"T0",["T","N"]

This call transforms the dictionary by creating the key *"T0"*. See Figure 2.15

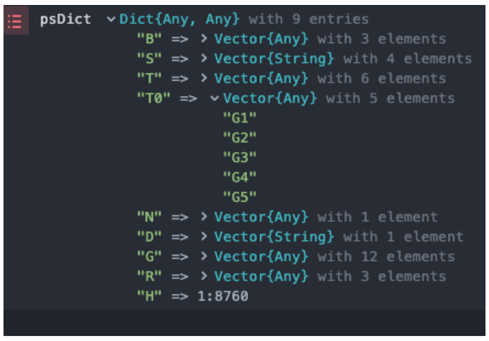
2.3.10 joinSets!

- Location: */functions/load_support.jl*
- Description: joins the elements of 2 sets and adds a new key to the set dictionary with these elements.



```
psDict  ⌄Dict{Any, Any} with 8 entries
            "B" => › Vector{Any} with 3 elements
            "S" => › Vector{String} with 4 elements
            "T" => ⌄Vector{Any} with 6 elements
                    "G1"
                    "G2"
                    "G3"
                    "G4"
                    "G5"
                    "G6"
            "N" => ⌄Vector{Any} with 1 element
                    "G6"
            "D" => › Vector{String} with 1 element
            "G" => › Vector{Any} with 12 elements
            "R" => › Vector{Any} with 3 elements
            "H" => 1:8760
```

(a) Set dictionary: before diffSets!



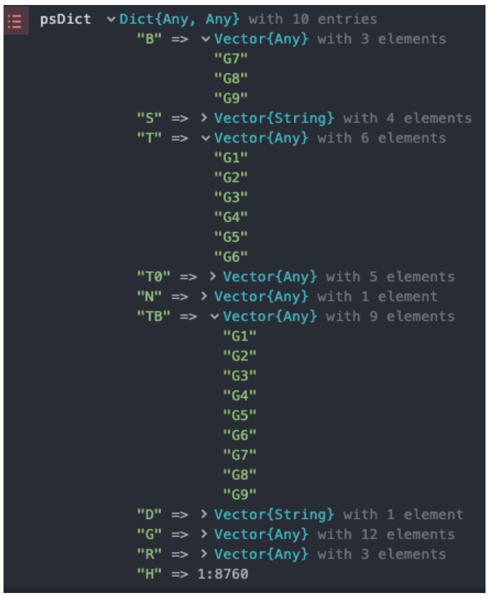
```
psDict  ⌄Dict{Any, Any} with 9 entries
            "B" => › Vector{Any} with 3 elements
            "S" => › Vector{String} with 4 elements
            "T" => › Vector{Any} with 6 elements
            "T0" => ⌄Vector{Any} with 5 elements
                     "G1"
                     "G2"
                     "G3"
                     "G4"
                     "G5"
            "N" => › Vector{Any} with 1 element
            "D" => › Vector{String} with 1 element
            "G" => › Vector{Any} with 12 elements
            "R" => › Vector{Any} with 3 elements
            "H" => 1:8760
```

(b) Set dictionary: after diffSets!

Figure 2.15: diffSets! function

- Usage: *joinSets!(dict::Dict, newset::String, sets::Vector{String})* as shown takes 3 arguments, transforming the first one, adding a new key entry with the elements in the sets present in the vector of sets.

This function is similar to diffSets! but instead of substracting the elements, it joins them. The small example is seen in Figure 2.16, where *"TB"* is the result of joining *"T"* and *"B"*.



```
psDict  ⌄Dict{Any, Any} with 10 entries
            "B" => ⌄Vector{Any} with 3 elements
                    "G7"
                    "G8"
                    "G9"
            "S" => › Vector{String} with 4 elements
            "T" => ⌄Vector{Any} with 6 elements
                    "G1"
                    "G2"
                    "G3"
                    "G4"
                    "G5"
                    "G6"
            "T0" => › Vector{Any} with 5 elements
            "N" => › Vector{Any} with 1 element
            "TB" => ⌄Vector{Any} with 9 elements
                     "G1"
                     "G2"
                     "G3"
                     "G4"
                     "G5"
                     "G6"
                     "G7"
                     "G8"
                     "G9"
            "D" => › Vector{String} with 1 element
            "G" => › Vector{Any} with 12 elements
            "R" => › Vector{Any} with 3 elements
            "H" => 1:8760
```

Figure 2.16: Set dictionary: after joinSets!

2.3.11 loadParameters

- Location: */functions/load_support.jl*
- Description: returns a dictionary with the parameters specified as in the *parameters* spreadsheet.
- Usage: *loadparameters(datawpath::Dict,paramSheet::String)::Dict* as shown takes only 2 arguments (dictionary at the level and the name of the table to read the parameters from; and returns a dictionary).

This function assumes a fix structure of the parameters table (dictionary and datasheet). The needed columns are explained here; first for a simple example, and later for a more complicated one.

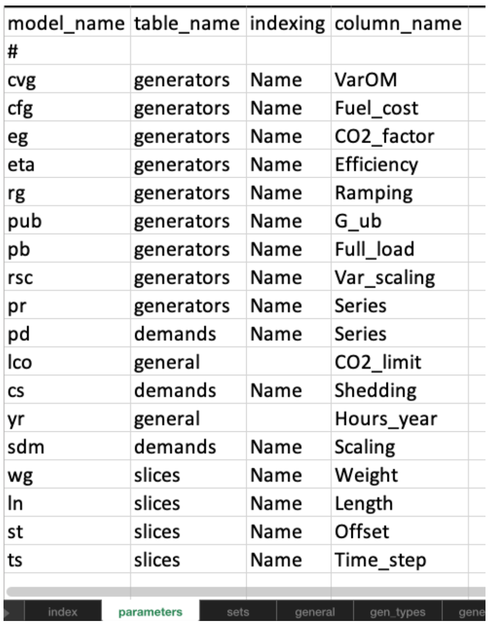
1. model_name: name of the parameter that will be created in Julia's memory.

2. table_name: table of elements which contains the list of elements where the parameter is going to be formed from.

3. indexing: indexing of the parameter, for example generator name.

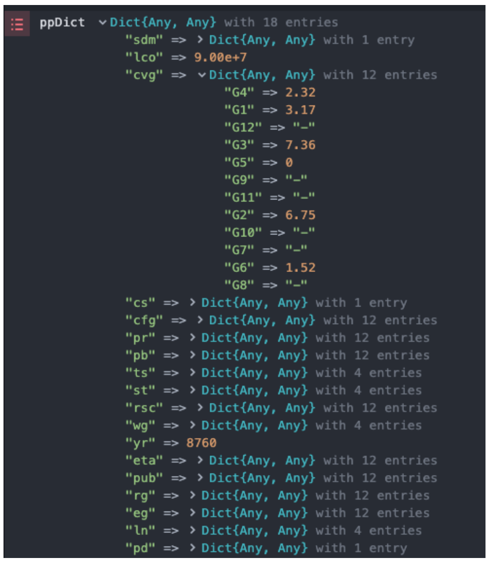
4. column_name: the column where the property is held.

In the Figure (2.17), there are single parameters such as *”lco”* which are not indexed, and othere indexed parameters such as *”cvg”*. In the figure, the input and output can be seen.



model_name	table_name	indexing	column_name
#			
cvg	generators	Name	VarOM
cfg	generators	Name	Fuel_cost
eg	generators	Name	CO2_factor
eta	generators	Name	Efficiency
rg	generators	Name	Ramping
pub	generators	Name	G_ub
pb	generators	Name	Full_load
rsc	generators	Name	Var_scaling
pr	generators	Name	Series
pd	demands	Name	Series
lco	general		CO2_limit
cs	demands	Name	Shedding
yr	general		Hours_year
sdm	demands	Name	Scaling
wg	slices	Name	Weight
ln	slices	Name	Length
st	slices	Name	Offset
ts	slices	Name	Time_step

(a) Data table: Parameters



```
ppDict  ⌄Dict{Any, Any} with 18 entries
          "sdm" => › Dict{Any, Any} with 1 entry
          "lco" => 9.00e+7
          "cvg" => ⌄Dict{Any, Any} with 12 entries
                      "G4" => 2.32
                      "G1" => 3.17
                      "G12" => "-"
                      "G3" => 7.36
                      "G5" => 0
                      "G9" => "-"
                      "G11" => "-"
                      "G2" => 6.75
                      "G10" => "-"
                      "G7" => "-"
                      "G6" => 1.52
                      "G8" => "-"
          "cs" => › Dict{Any, Any} with 1 entry
          "cfg" => › Dict{Any, Any} with 12 entries
          "pr" => › Dict{Any, Any} with 12 entries
          "pb" => › Dict{Any, Any} with 12 entries
          "ts" => › Dict{Any, Any} with 4 entries
          "st" => › Dict{Any, Any} with 4 entries
          "rsc" => › Dict{Any, Any} with 12 entries
          "wg" => › Dict{Any, Any} with 4 entries
          "yr" => 8760
          "eta" => › Dict{Any, Any} with 12 entries
          "pub" => › Dict{Any, Any} with 12 entries
          "rg" => › Dict{Any, Any} with 12 entries
          "eg" => › Dict{Any, Any} with 12 entries
          "ln" => › Dict{Any, Any} with 4 entries
          "pd" => › Dict{Any, Any} with 1 entry
```

(b) Output dictionary: Parameters

Figure 2.17: loadParameters function

The second “more” complicated example comes from a the same instance (Nicolo/dataXLSX) but for the investment part, as many of the properties need to be indexed by element (e.g. generator) and investment or operational node in the tree.

In Figure 2.18a the table used for loadParameters! is shown for the investment problem; Figure 2.18b shows the . In there, there is not only one table_name index, but the format is table_name_# where “#” represents any character. For simplicity, they have been chosen as 1 and 2. The 3 different structured parameters are “dsc”, “xh” and “ci”.

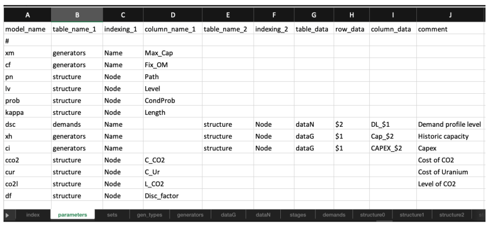
The first indexing dimension (subindex 1) has demands or generators, depending on the parameters that have been indexed by name. Then in the second level the parameters are indexed by Node coming from the structure table. See Figure 2.19a. Addtional, as there are 2 indices, the data that feeds the parameters is found on a different table (dataG for xh and ci, and dataN for dsc). There are 3 additional columns for these cases: “table_data”, “row_data” and “column_data”. “table_data” as previously explained, refers to the table where the data is held, then “row_data” indicates how it is indexed in the rows (in this example) using the same subindex (1 or 2), and then the “column_data” shows the use of a string put together with the other index. This explanation is much clearer when seeing the Figures 2.19b and 2.19c.

2.3.12 augmentStructure

- Location: */functions/load_support.jl*

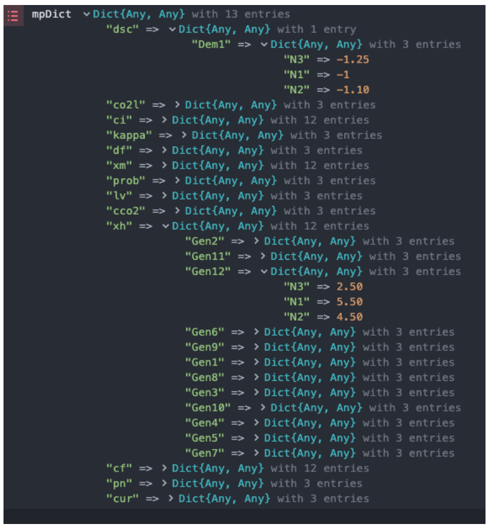
- Description: adds auxiliary entries to the structure table to help with model building.

- Usage: *augmentStructure(dict::Dict, structureTable::String* as shown only takes 2 arguments, transforming the first one, adding a new key entries with the elements of the structure.



A	B	C	D	E	F	G	H	I	J
model_name	table_name_1	indexing_1	column_name_1	table_name_2	indexing_2	table_data	row_data	column_data	comment
#									
xm	generators	Name	Max_Cap						
cf	generators	Name	Fix_OM						
pn	structure	Node	Path						
lv	structure	Node	Level						
prob	structure	Node	CondProb						
kappa	structure	Node	Length						
dsc	demands	Name		structure	Node	dataN	$2	DL_$1	Demand profile level
xh	generators	Name		structure	Node	dataG	$1	Cap_$2	Historic capacity
ci	generators	Name		structure	Node	dataG	$1	CAPEX_$2	Capex
cco2	structure	Node	C_CO2						Cost of CO2
cur	structure	Node	C_Ur						Cost of Uranium
co2l	structure	Node	L_CO2						Level of CO2
df	structure	Node	Disc_factor						

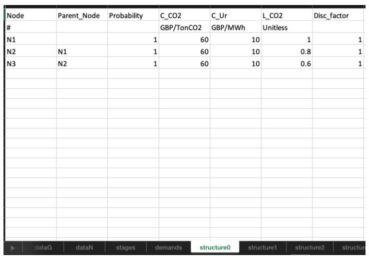
(a) Data table: Investment Parameters (2 indices)



```
mpDict  ⌄Dict{Any, Any} with 13 entries
          "dsc" => ⌄Dict{Any, Any} with 1 entry
                    "Dem1" => ⌄Dict{Any, Any} with 3 entries
                                "N3" => -1.25
                                "N1" => -1
                                "N2" => -1.10
          "co2l" => › Dict{Any, Any} with 3 entries
          "ci" => › Dict{Any, Any} with 12 entries
          "kappa" => › Dict{Any, Any} with 3 entries
          "df" => › Dict{Any, Any} with 3 entries
          "xm" => › Dict{Any, Any} with 12 entries
          "prob" => › Dict{Any, Any} with 3 entries
          "lv" => › Dict{Any, Any} with 3 entries
          "cco2" => › Dict{Any, Any} with 3 entries
          "xh" => ⌄Dict{Any, Any} with 12 entries
                   "Gen2" => › Dict{Any, Any} with 3 entries
                   "Gen11" => › Dict{Any, Any} with 3 entries
                   "Gen12" => ⌄Dict{Any, Any} with 3 entries
                                "N3" => 2.50
                                "N1" => 5.50
                                "N2" => 4.50
                   "Gen6" => › Dict{Any, Any} with 3 entries
                   "Gen9" => › Dict{Any, Any} with 3 entries
                   "Gen1" => › Dict{Any, Any} with 3 entries
                   "Gen8" => › Dict{Any, Any} with 3 entries
                   "Gen3" => › Dict{Any, Any} with 3 entries
                   "Gen10" => › Dict{Any, Any} with 3 entries
                   "Gen4" => › Dict{Any, Any} with 3 entries
                   "Gen5" => › Dict{Any, Any} with 3 entries
                   "Gen7" => › Dict{Any, Any} with 3 entries
          "cf" => › Dict{Any, Any} with 12 entries
          "pn" => › Dict{Any, Any} with 3 entries
          "cur" => › Dict{Any, Any} with 3 entries
```

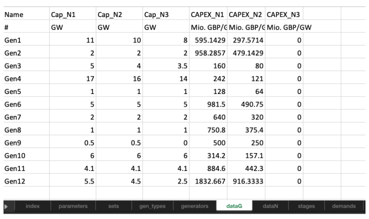
(b) Output dictionary: Investment Parameters

Figure 2.18: loadParameters function (with 2 indices)



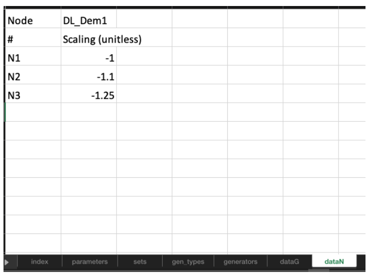
Node	Parent_Node	Probability	C_CO2	C_Ur	L_CO2	Disc_factor
#			GBP/TonCO2	GBP/MWh	Unitless	
N1		1	60	10	1	1
N2	N1	1	60	10	0.8	1
N3	N2	1	60	10	0.6	1

(a) Structure table: Nodes



Name	Cap_N1	Cap_N2	Cap_N3	CAPEX_N1	CAPEX_N2	CAPEX_N3
#	GW	GW	GW	Mio. GBP/G	Mio. GBP/G	Mio. GBP/GW
Gen1	11	10	8	595.1429	297.5714	0
Gen2	2	2	2	958.2857	479.1429	0
Gen3	5	4	3.5	160	80	0
Gen4	17	16	14	242	121	0
Gen5	1	1	1	128	64	0
Gen6	5	5	5	981.5	490.75	0
Gen7	2	2	2	640	320	0
Gen8	1	1	1	750.8	375.4	0
Gen9	0.5	0.5	0	500	250	0
Gen10	6	6	6	314.2	157.1	0
Gen11	4.1	4.1	4.1	884.6	442.3	0
Gen12	5.5	4.5	2.5	1832.667	916.3333	0

(b) Table: dataG



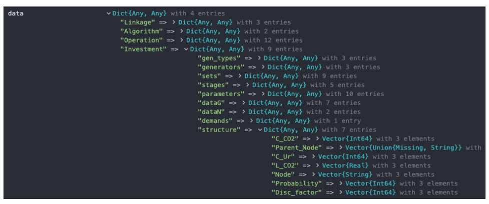
Node	DL_Dem1
#	Scaling (unitless)
N1	-1
N2	-1.1
N3	-1.25

(c) Table: dataN

Figure 2.19: Additional parameter tables

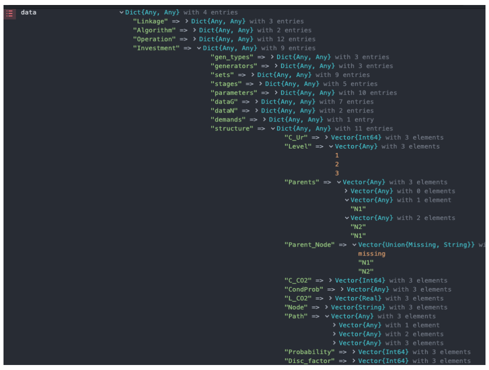
The function adds information inferred from the "structure" table. The newly added information is: "CondProb" which refers to the global probability in the node, based on its parents probability. "Level" refers to the stage of the node, for example in N3 happens after N2, and N2 is the child of N1, the stage of this node would be 3. "Path" calculates all the parents to the root node. "Parents" refers to the immediate parent of a node.

The dictionary before and calling the function can be seen in Figure 2.20



```
data                ⌄Dict{Any, Any} with 4 entries
                      "Linkage" => › Dict{Any, Any} with 3 entries
                      "Algorithm" => › Dict{Any, Any} with 2 entries
                      "Operation" => › Dict{Any, Any} with 12 entries
                      "Investment" => ⌄Dict{Any, Any} with 9 entries
                                         "gen_types" => › Dict{Any, Any} with 3 entries
                                         "generators" => › Dict{Any, Any} with 3 entries
                                         "sets" => › Dict{Any, Any} with 9 entries
                                         "stages" => › Dict{Any, Any} with 5 entries
                                         "parameters" => › Dict{Any, Any} with 10 entries
                                         "dataG" => › Dict{Any, Any} with 7 entries
                                         "dataN" => › Dict{Any, Any} with 2 entries
                                         "demands" => › Dict{Any, Any} with 1 entry
                                         "structure" => ⌄Dict{Any, Any} with 7 entries
                                                          "C_CO2" => › Vector{Int64} with 3 elements
                                                          "Parent_Node" => › Vector{Union{Missing, String}} with
                                                          "C_Ur" => › Vector{Int64} with 3 elements
                                                          "L_CO2" => › Vector{Real} with 3 elements
                                                          "Node" => › Vector{String} with 3 elements
                                                          "Probability" => › Vector{Int64} with 3 elements
                                                          "Disc_factor" => › Vector{Int64} with 3 elements
```

(a) Structure table: before call



```
data                ⌄Dict{Any, Any} with 4 entries
                      "Linkage" => › Dict{Any, Any} with 3 entries
                      "Algorithm" => › Dict{Any, Any} with 2 entries
                      "Operation" => › Dict{Any, Any} with 12 entries
                      "Investment" => ⌄Dict{Any, Any} with 9 entries
                                         "gen_types" => › Dict{Any, Any} with 3 entries
                                         "generators" => › Dict{Any, Any} with 3 entries
                                         "sets" => › Dict{Any, Any} with 9 entries
                                         "stages" => › Dict{Any, Any} with 5 entries
                                         "parameters" => › Dict{Any, Any} with 10 entries
                                         "dataG" => › Dict{Any, Any} with 7 entries
                                         "dataN" => › Dict{Any, Any} with 2 entries
                                         "demands" => › Dict{Any, Any} with 1 entry
                                         "structure" => ⌄Dict{Any, Any} with 11 entries
                                                          "C_Ur" => › Vector{Int64} with 3 elements
                                                          "Level" => ⌄Vector{Any} with 3 elements
                                                                      1
                                                                      2
                                                                      3
                                                          "Parents" => ⌄Vector{Any} with 3 elements
                                                                         › Vector{Any} with 0 elements
                                                                         ⌄Vector{Any} with 1 element
                                                                          "N1"
                                                                         ⌄Vector{Any} with 2 elements
                                                                          "N2"
                                                                          "N1"
                                                          "Parent_Node" => ⌄Vector{Union{Missing, String}} wi
                                                                             missing
                                                                             "N1"
                                                                             "N2"
                                                          "C_CO2" => › Vector{Int64} with 3 elements
                                                          "CondProb" => › Vector{Any} with 3 elements
                                                          "L_CO2" => › Vector{Real} with 3 elements
                                                          "Node" => › Vector{String} with 3 elements
                                                          "Path" => ⌄Vector{Any} with 3 elements
                                                                      › Vector{Any} with 1 element
                                                                      › Vector{Any} with 2 elements
                                                                      › Vector{Any} with 3 elements
                                                          "Probability" => › Vector{Int64} with 3 elements
                                                          "Disc_factor" => › Vector{Int64} with 3 elements
```

(b) Structure table: after call

Figure 2.20: augmentStructure function

2.3.13 addMapTable

- Location: */functions/load_support.jl*
- Description: adds auxiliary entries to the structure table to help with model building.
- Usage: *addMapTable(dict::Dict, structureTable::String, nodeString::String* as shown only takes 3 arguments, transforming the first one, adding a new key entries with the elements of the structure and node information.

This is a custom function that is helpful when building the particular Nicolo/dataXLSX case. It adds a key that holds the of parents which are investment and operation relevant (these are subsets of the Path lists created with augmentStructure).

The idea of this function is to allow for only operation nodes to be added which the last investment node is not exactly the preceding one. This function is better understood when looking at the function source code together with Nicolo/dataXLSX example.

2.3.14 transformStructure

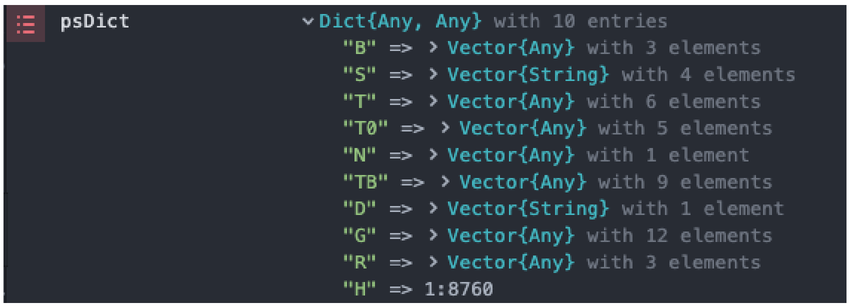
- Location: */functions/load_support.jl*
- Description: takes a dictionary and returns a Julia (mutable) structure.
- Usage: *transformStructure(dict::Dict, structNameType::String*, the first argument is the dictionary to be transformed, then the name (type) of the mutable structure.

The function is written using the Metaprogramming modules from Julia. It takes a dictionary and transforms each of its first level keys into components of the structure. A structure is a much more eficient (and has shorter calls) than a dictionary. It also matches the information expected by the original part of the planner coded for the paper.

In the example, the function is called and its output is stored in a variable called “ps” of type “ps_type” (structure).

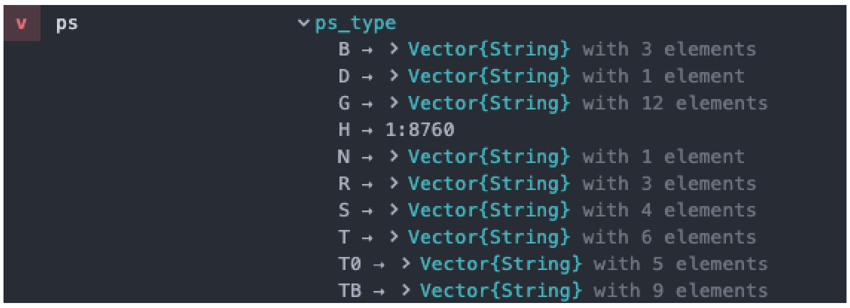
ps=transformStructure(psDict,”ps”)

Dictionary psDict is transformed into ps. Figure 2.21 shows clearly the input dictionary and the output structure.



```
psDict                  Dict{Any, Any} with 10 entries
                          "B" => › Vector{Any} with 3 elements
                          "S" => › Vector{String} with 4 elements
                          "T" => › Vector{Any} with 6 elements
                          "T0" => › Vector{Any} with 5 elements
                          "N" => › Vector{Any} with 1 element
                          "TB" => › Vector{Any} with 9 elements
                          "D" => › Vector{String} with 1 element
                          "G" => › Vector{Any} with 12 elements
                          "R" => › Vector{Any} with 3 elements
                          "H" => 1:8760
```

(a) Input dictionary



```
v ps                    ps_type
                          B → › Vector{String} with 3 elements
                          D → › Vector{String} with 1 element
                          G → › Vector{String} with 12 elements
                          H → 1:8760
                          N → › Vector{String} with 1 element
                          R → › Vector{String} with 3 elements
                          S → › Vector{String} with 4 elements
                          T → › Vector{String} with 6 elements
                          T0 → › Vector{String} with 5 elements
                          TB → › Vector{String} with 9 elements
```

(b) Output structure

Figure 2.21: transformStructure function

Chapter 3

Optimization Models

In the planner, there are options to solve problems using different methodologies. The problem can be solved undecomposed (as a single model), or using decomposition: either Standard Benders or Adaptive Benders. If the model is solved undecomposed, only an undecomposed model object is needed. With decomposition 2 separate models are needed: The operational model and the investment planning model. In this planner, the investment problem is the master problem and the operational problems are the subproblems. In case stabilisation is activated for the decompositions, a third model based on the master problem will also be created.

The output of the data manager (Chapter 2) is 4 Julia structures:

1. ps (ps_type): sets used in the operational problem (subproblem)
2. pp (pp_type): parameters used in the operational problem (subproblem)
3. ms (ms_type): sets used in the investment planning problem (master problem)
4. mp (mp_type): parameters used in the investment planning problem (master problem)

The data contained in these structures will be used for model building purposes as it will be explained in this section.

3.1 Model building

The model objects are created in the "/functions/link_models.jl" script. First, and empty model object is created with the JuMP function *Model()*, and the solver and starting attributes are chosen. In Section 6.2, line by line explanation will be given.

An example of the subproblem creation and population is shown with the following function call:

m = Model(optimizer_with_attributes(()->Gurobi.Optimizer(),"Method"=>2)

In this example, an empty model object called *m* is created and assigned to be solved by Gurobi with the "Method" attribute of 2, which according to Gurobi's documentation corresponds to interior point solution method.

Later the models are populated with custom functions called *SP!*, *RMP!* or *Undecomposed!*.

3.2 Model definition

As the methodology used by the Planner is based on Benders decomposition, the problem has to be divided into a Relaxed Master Problem and a Subproblem. A nice and practical overview of this method can be found in [2].

3.2.1 Undecomposed problem

First, let's talk about the original (undecomposed) model. This is the target instance, where everything is solved as a single model, and everything is handled directly by the solver without the need of Benders decomposition.

For small problems, the undecomposed version will tend to outperform decomposition. In large problems, the performance of the decomposition will depend on the instance, depending on how much can the block-diagonal structure be exploited.

The undecomposed problem exists in the planner to help with benchmarks.

The form of the (undecomposed) problem is:

The specific model depends on the instance to be solved. An explicit example will be given in Chapter 7.

3.2.2 Master Problem

The master problem in the planner corresponds to the Investment part. The example objective used for the examples that accompany the software optimize the investment in capacity of different elements (generators and lines) such that the total operational and investment cost is minimised.

In general, the form of the master problem is:

The specific model depends on the instance to be solved. An explicit example will be given in Chapter 7.

3.2.3 Subproblem

The subproblem in the planner corresponds to the operational problem, i.e. what is the operational cost for a given investment.

The way that Benders decomposition works is that the Master problem will fix some variables in the operation, and operation will be optimized returning to the master problem some derivative information and the objective value for a given investment.

The form of the subproblems are:

The specific model depends on the instance to be solved. An explicit example will be given in Chapter 7.

3.3 Functions

Regarding the models, there are only 3 functions that help their creation. The information for building them, sets and parameters, comes from the data manager chapter of the planner.

3.3.1 SP!

- Location: */$DataPath/load_models.jl*
- Description: creates the subproblem model object
- Usage: *SP!(m,ps,pp)* as shown takes 3 arguments - m refers to the model object that is to be populated, ps refers to the operational sets and pp to the operational parameters. The function returns the modified (populated) m object.

3.3.2 RMP!

- Location: */$DataPath/load_models.jl*
- Description: creates the master problem model object
- Usage: *RMP!(m,ms,mp)* as shown takes 3 arguments - m refers to the model object that is to be populated, ms refers to the investment sets and mp to the investment parameters. The function returns the modified (populated) m object.

3.3.3 Undecomposed!

- Location: */$DataPath/load_models.jl*
- Description: creates the undecomposed problem model object
- Usage: *Undecomposed!(m,ms,mp,ps,pp,lT)* as shown takes 6 arguments - *m* refers to the model object that is to be populated, *ms* refers to the sets, *mp* to the parameters and *lT* to a linking table. The function returns the modified (populated) *m* object.

This particular function requires a linking table because is supossed to be a 1-to-1 match to the decomposition. It uses the linking table to "connect" the models. In reality for an undecomposed model it is not needed, but recommended to be sure that the model is correctly linked and the optimal value matches the one given by the decompositions.

Alternatively, the function could be written with 5 arguments (*m*,*mp*,*sp*,*pp*) used for the other models, but it wouldn't be a 1-to-1 match to the decomposition. The location of the file is inside $DataPath which means for the example, inside *Nicolo*.

3.3.4 oCost

- Location: */$DataPath/load_models.jl*
- Description: defines the objective of the operational model as a function
- Usage: *oCost(mp,beta,i)* the arguments vary depending on what is needed. In the Nicolo example, the objective function is the total cost of operation.

The need of the *oCost* function stems from the fact that in different places the operational cost can queried as an exact objective or an approximation on the objective (such as with the oracles in Adaptive Benders) so having a function gives the flexibility that is needed so the bounds are calculated correctly down the process.

3.3.5 updateObjLMP!

- Location: */$DataPath/load_models.jl*
- Description: sets the objective function of the stabilisation problem
- Usage: *updateObjLMP!(...)* the arguments vary depending on what is needed. In the Nicolo example, the objective function could be a zero function or a specific norm minimisation problem for stabilisation.

This is only needed when stabilisation is activated, and helps the user to set the stabilisation strategy of the objective function.

3.3.6 auxData

- Location: */$DataPath/load_models.jl*
- Description: helps retrieve data from arrays to dictionaries.
- Usage: *auxData(...)* the arguments vary depending on what is needed. In the Nicolo example, the stabilisation is done with respect to the variable **:pN**.

The stabilisation can be done on different variables, and this function helps bridging the form of the arrays (the planner is matrix based), while for flexibility reasons the models are written in terms of dictionaries.

Chapter 4

Model linker

As shown in the previous chapter, the models are created separately by the SP! and RMP! functions. In order for Benders to work, they need to be linked and augmented with some coordinating variables. To link the models, first a table with specific structure needs to be available (already read by the data reader module); and then, tailor functions will create a dictionary that will be used for linking the models.

4.0.1 Linking Tables

The XLSX version of the data includes a file called *link.xlsx* (see Figure 2.1), apart from the index sheet that exist for the data reader. The specific tables (sheets) are: indices, variables and definitions.

Definitions Table

The definitions table holds a special structure. The definitions are:

1. model_name: has the name of the objects created and populated by the corresponding RMP! and SP! functions.
2. set_container: has the name of the sets for both operational and investment
3. parameter_container: similar to set_container, but for parameters
4. model_type: adds a name of operation or investment (currently not used, but do not remove)

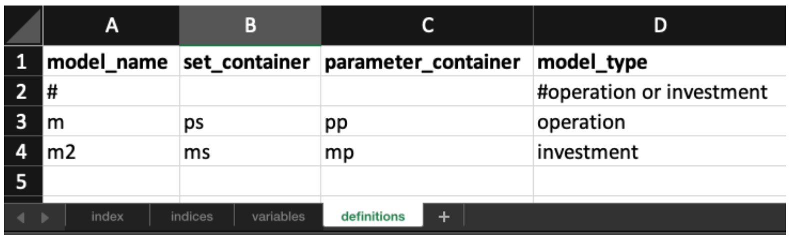
See Figure 4.1 with example data for the definitions table.

Variables Table

The variables table holds a special structure. The names of the variables and the information about which sets from investment should be linked to other sets in operation resides here.

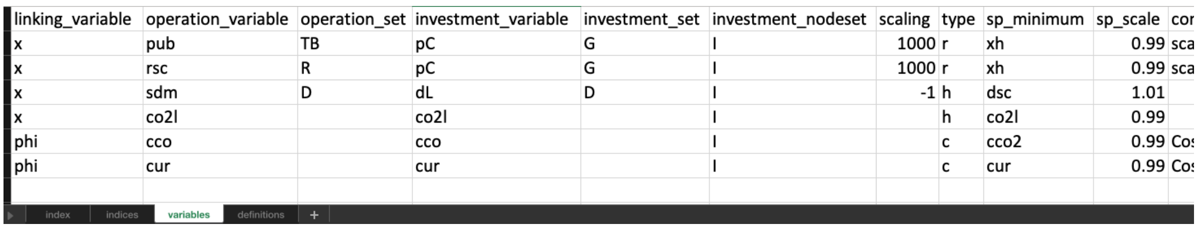
The columns in the variables table are:

1. linking_variable: there are two types of linking variable values allowed - **x** and **phi**. **x** is used for right-hand sides that need to be coordinated by Benders, for example cumulative capacity passed to the subproblem or or other exogenous parameters carbon scaling, demands scaling, etcetera. **phi** refers to cost coefficients that are changing in different nodes in the tree, for example the cost of coal or the CO2 tax.



	A	B	C	D
1	**model_name**	**set_container**	**parameter_container**	**model_type**
2	#			#operation or investment
3	m	ps	pp	operation
4	m2	ms	mp	investment
5				

Figure 4.1: definitions sheet in link.xlsx



linking_variable	operation_variable	operation_set	investment_variable	investment_set	investment_nodeset	scaling	type	sp_minimum	sp_scale	cor
x	pub	TB	pC	G	I	1000	r	xh	0.99	sca
x	rsc	R	pC	G	I	1000	r	xh	0.99	sca
x	sdm	D	dL	D	I	-1	h	dsc	1.01	
x	co2l		co2l		I		h	co2l	0.99	
phi	cco		cco		I		c	cco2	0.99	Co
phi	cur		cur		I		c	cur	0.99	Co

Figure 4.2: variables sheet in link.xlsx

2. operation_variable: refers to the name of the coordinated (fixed) variable in the operational model. For example pG_{ub} in operation means the installed capacity.

3. operation_set: refers to the set, over which the operation variable is defined.

4. investment_variable: refers to the name of the variable in the master problem (investment) that is to be passed to the subproblem.

5. investment_set: includes the set over which the investment variable is defined.

6. investment_nodeset: for investment problems, includes the investment set, that includes the nodes where the decisions for investment can take place.

7. scaling: if there is a mismatch between master problem passed variables and operational problem received variables, this will be set here. In the Nicolo's example, the RMP (investment problem) refers to investment variables in GW, whereas the SP (operational problem) uses as basis MW. So the 1000.0 scaling allows the linking and each model to be performed

8. type: there are three types used in this field - **r** means real variable in the investment problem which is then fixed as a parameter in the operational problem, **h** is used for exogenous parameters in both the investment and operational levels; **c** means that it is a cost coefficient. By force, there should be a match between linking_variable and type, such that **x** can only combine with **r** and **h**, and **phi** with **c**.

9. sp_minimum: the way the adaptive Benders work needs to include a special point, that helps forming the convex combination of the lower bound oracle to be always feasible. The special point normally corresponds the the minimum investment values that the problem can have. In the example, the minimum historical capacity of generation is used *xh* in the table. The values on these cells need are referenced to an investment parameter defined over the same set as the investment variable.

10. sp_scale: sometimes for numerical stability purposes, the special point does not correspond to the exact minimum of a parameter, instead is offset slightly (like in this examples 0.99).

See Figure 4.2 with example data for the variables table.

Indices Table

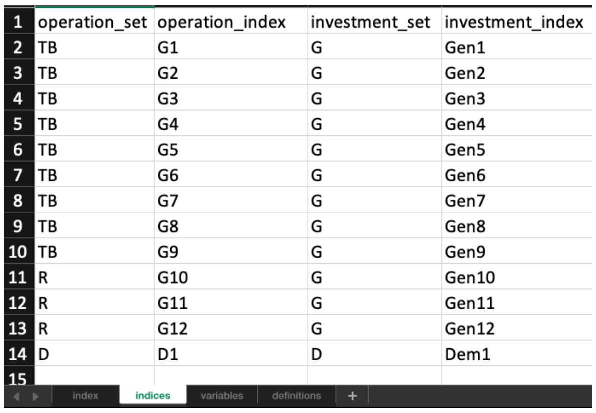
This table corresponds the explicit match between operational and investment sets and parameters.
See Figure 4.3 with example data for the variables table.

4.1 Functions

4.1.1 linkingTable

- Location: */functions/load_support.jl*

- Description: after reading the relevant data, it creates a dictionary for the linking table and a map between node index and node name for investment.



	operation_set	operation_index	investment_set	investment_index
2	TB	G1	G	Gen1
3	TB	G2	G	Gen2
4	TB	G3	G	Gen3
5	TB	G4	G	Gen4
6	TB	G5	G	Gen5
7	TB	G6	G	Gen6
8	TB	G7	G	Gen7
9	TB	G8	G	Gen8
10	TB	G9	G	Gen9
11	R	G10	G	Gen10
12	R	G11	G	Gen11
13	R	G12	G	Gen12
14	D	D1	D	Dem1
15				

index
indices
variables
definitions
+

Figure 4.3: indices sheet in link.xlsx

- Usage: *linkingTable(datawpath::Dict)* as shown only takes the dictionary with the relevant linking information, which was loaded by the data loader module and the specific format of the tables presented in this chapter. It returns a tuple of objects, the first one is the linkingTable dictionary which contains a map of each one of the variables that needs to be coordinated between problems, the second dictionary is the map between node index to node name.

To understand the outputs, in this example called *lT* and *i2n*, it is convenient to relate Figures 4.1, 4.2 and 4.3, and try relating them to the Figure 4.4.

The dictionary in Figure 4.4 shows that each one of the variables mentioned in the variables sheet presented earlier in this chapter is expanded in the sets of the elements with their respective indices. The Figure shows the first 2 variables for the cumulative capacity with all the information of the names of model, names of variables and indices. It also shows the scaling and the "special point" to be evaluated.

Figure 4.5 simply shows the index of an investment node against its index that will be used inside the Planner (solver) part.

4.1.2 addLinking!

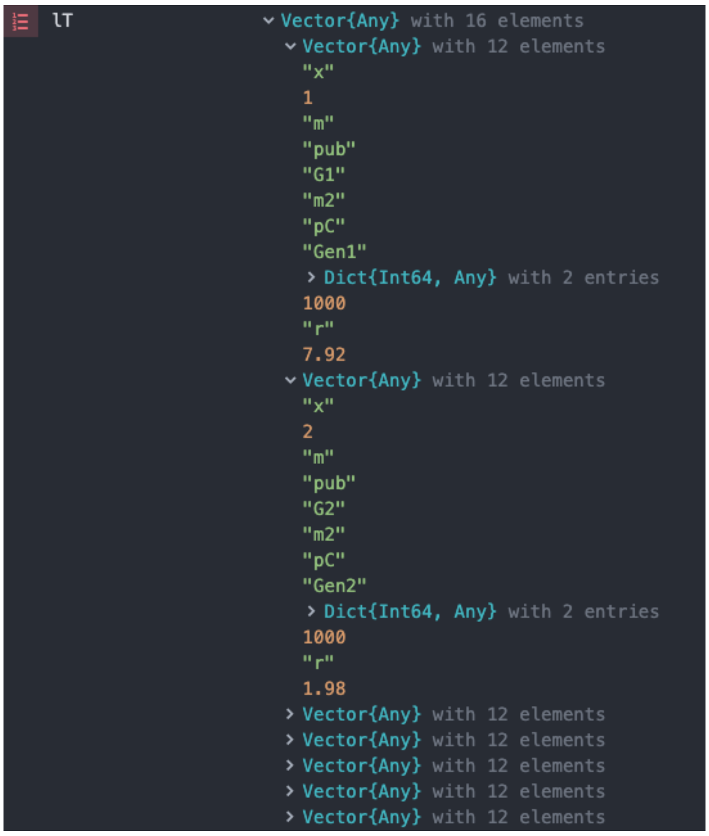
- Location: */functions/load_support.jl*
- Description: the function modifies the models (master and subproblem) by adding the needed variables for coordinating them. It uses the meta-programming framework of Julia to read from the linking table and add elements to the JuMP models.
- Usage: *addLinking!(linkingTable::Dict)* as shown only takes the dictionary for the linking table, created with the function *linkingTable* and reads the models and augments them.

When the operational model was created (and matching the info in the definitions table) the operational model object was given the name **m**, and the investment one the name **m2**. The Figures 4.6a, 4.6b and 4.6e show the changes to the operational model. Figure 4.6a shows the number of variables, constraints and the names registered in the model. Once the function is called, these model gets modified and then 4.6b shows that the model is larger now, because new variables, constraints and names were introduced for coordination. Finally, Figure 4.6e shows some of the newly introduced constraints and variables, so the user can have a feel for the changes.

Similar to Figures 4.6a, 4.6b and 4.6e, Figures 4.6c, 4.6d and 4.6f show the analogous process for the investment problem (the object is called **m2**).

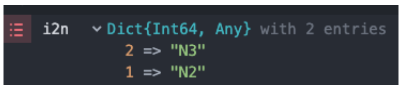
4.1.3 writeUNC

- Location: */functions/load_support.jl*
- Description: returns a structure for the "uncertainty" part needed for the planner. Given a set of investment nodes, operation nodes and names, it will sort the structure uncertainties, creating matrices for different right-hand sides and cost coefficients in the specific order requested by the planner.



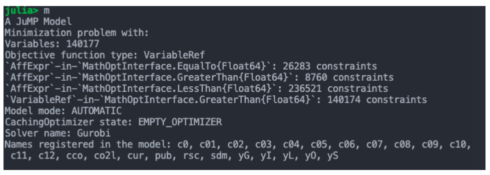
```
lT    ⌄ Vector{Any} with 16 elements
        ⌄ Vector{Any} with 12 elements
            "x"
            1
            "m"
            "pub"
            "G1"
            "m2"
            "pC"
            "Gen1"
            › Dict{Int64, Any} with 2 entries
            1000
            "r"
            7.92
        ⌄ Vector{Any} with 12 elements
            "x"
            2
            "m"
            "pub"
            "G2"
            "m2"
            "pC"
            "Gen2"
            › Dict{Int64, Any} with 2 entries
            1000
            "r"
            1.98
        › Vector{Any} with 12 elements
        › Vector{Any} with 12 elements
        › Vector{Any} with 12 elements
        › Vector{Any} with 12 elements
        › Vector{Any} with 12 elements
```

Figure 4.4: Linking Table



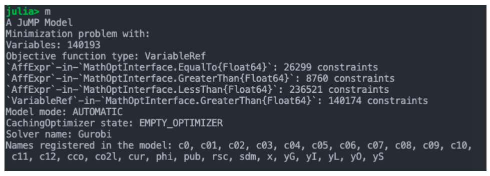
```
i2n   ⌄ Dict{Int64, Any} with 2 entries
          2 => "N3"
          1 => "N2"
```

Figure 4.5: index to node map



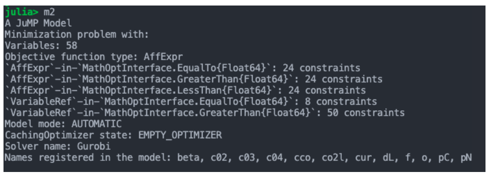
```
julia> m
A JuMP Model
Minimization problem with:
Variables: 140177
Objective function type: VariableRef
`AffExpr`-in-`MathOptInterface.EqualTo{Float64}`: 26283 constraints
`AffExpr`-in-`MathOptInterface.GreaterThan{Float64}`: 8760 constraints
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 236521 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 140174 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: Gurobi
Names registered in the model: c0, c01, c02, c03, c04, c05, c06, c07, c08, c09, c10,
 c11, c12, cco, co2l, cur, pub, rsc, sdm, yG, yI, yL, yO, yS
```

(a) Operational model before addLinking!



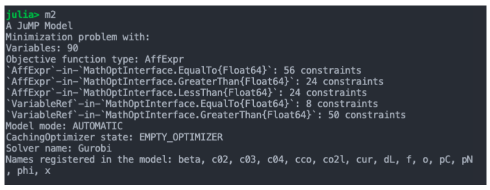
```
julia> m
A JuMP Model
Minimization problem with:
Variables: 140193
Objective function type: VariableRef
`AffExpr`-in-`MathOptInterface.EqualTo{Float64}`: 26299 constraints
`AffExpr`-in-`MathOptInterface.GreaterThan{Float64}`: 8760 constraints
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 236521 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 140174 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: Gurobi
Names registered in the model: c0, c01, c02, c03, c04, c05, c06, c07, c08, c09, c10,
 c11, c12, cco, co2l, cur, phi, pub, rsc, sdm, x, yG, yI, yL, yO, yS
```

(b) Operational model after addLinking!



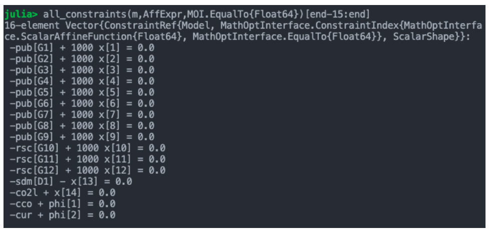
```
julia> m2
A JuMP Model
Minimization problem with:
Variables: 58
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.EqualTo{Float64}`: 24 constraints
`AffExpr`-in-`MathOptInterface.GreaterThan{Float64}`: 24 constraints
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 24 constraints
`VariableRef`-in-`MathOptInterface.EqualTo{Float64}`: 8 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 50 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: Gurobi
Names registered in the model: beta, c02, c03, c04, cco, co2l, cur, dL, f, o, pC, pN
```

(c) Investment model before addLinking!



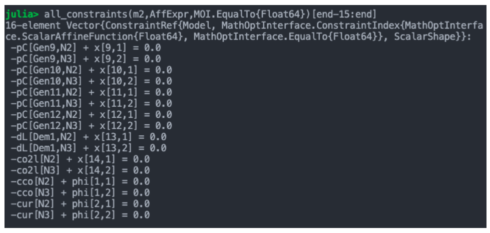
```
julia> m2
A JuMP Model
Minimization problem with:
Variables: 90
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.EqualTo{Float64}`: 56 constraints
`AffExpr`-in-`MathOptInterface.GreaterThan{Float64}`: 24 constraints
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 24 constraints
`VariableRef`-in-`MathOptInterface.EqualTo{Float64}`: 8 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 50 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: Gurobi
Names registered in the model: beta, c02, c03, c04, cco, co2l, cur, dL, f, o, pC, pN
, phi, x
```

(d) Investment model after addLinking!



```
julia> all_constraints(m,AffExpr,MOI.EqualTo{Float64})[end-15:end]
16-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterfa
ce.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 -pub[G1] + 1000 x[1] = 0.0
 -pub[G2] + 1000 x[2] = 0.0
 -pub[G3] + 1000 x[3] = 0.0
 -pub[G4] + 1000 x[4] = 0.0
 -pub[G5] + 1000 x[5] = 0.0
 -pub[G6] + 1000 x[6] = 0.0
 -pub[G7] + 1000 x[7] = 0.0
 -pub[G8] + 1000 x[8] = 0.0
 -pub[G9] + 1000 x[9] = 0.0
 -rsc[G10] + 1000 x[10] = 0.0
 -rsc[G11] + 1000 x[11] = 0.0
 -rsc[G12] + 1000 x[12] = 0.0
 -sdm[D1] - x[13] = 0.0
 -co2l + x[14] = 0.0
 -cco + phi[1] = 0.0
 -cur + phi[2] = 0.0
```

(e) Newly added constraints to operation (extract)



```
julia> all_constraints(m2,AffExpr,MOI.EqualTo{Float64})[end-15:end]
16-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterfa
ce.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 -pC[Gen9,N2] + x[9,1] = 0.0
 -pC[Gen9,N3] + x[9,2] = 0.0
 -pC[Gen10,N2] + x[10,1] = 0.0
 -pC[Gen10,N3] + x[10,2] = 0.0
 -pC[Gen11,N2] + x[11,1] = 0.0
 -pC[Gen11,N3] + x[11,2] = 0.0
 -pC[Gen12,N2] + x[12,1] = 0.0
 -pC[Gen12,N3] + x[12,2] = 0.0
 -dL[Dem1,N2] + x[13,1] = 0.0
 -dL[Dem1,N3] + x[13,2] = 0.0
 -co2l[N2] + x[14,1] = 0.0
 -co2l[N3] + x[14,2] = 0.0
 -cco[N2] + phi[1,1] = 0.0
 -cco[N3] + phi[1,2] = 0.0
 -cur[N2] + phi[2,1] = 0.0
 -cur[N3] + phi[2,2] = 0.0
```

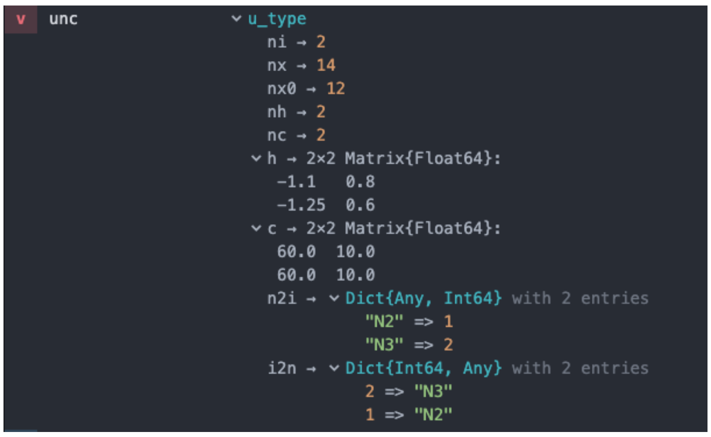
(f) Newly added constraints to investment (extract)

Figure 4.6: Operational/Investment model modifications to allow correct linking

- Usage: *writeUNC(lT,"u",n2i,i2n)* as shown takes 4 arguments, the linking table generated by the fucntion *linkingTable*, a string to define the type of the structure, and the the node-to-index and index-to-node dictionaries.

After running the function, it will return a new structure. In Figure 4.7 the contents can be seen.

In the structure, **ni** shows the number of investment nodes, **nx** the number of "x" variables, **nx0** the number of true "x" variables (removing exogenous RHS parameters from the list), **nh** is the number of exogenous RHS parameters, **nc** is the number of cost coefficients. **h** and **c** are the matrices for the investment nodes for the exogenous parameters and cost coefficients. **n2i** and **i2n** are similar to the input parameters of the function i.e. the node-to-index and index-to-node maps.



```
v  unc                 ⌄ u_type
                          ni → 2
                          nx → 14
                          nx0 → 12
                          nh → 2
                          nc → 2
                        ⌄ h → 2×2 Matrix{Float64}:
                           -1.1   0.8
                           -1.25  0.6
                        ⌄ c → 2×2 Matrix{Float64}:
                           60.0  10.0
                           60.0  10.0
                          n2i → ⌄ Dict{Any, Int64} with 2 entries
                                  "N2" => 1
                                  "N3" => 2
                          i2n → ⌄ Dict{Int64, Any} with 2 entries
                                  2 => "N3"
                                  1 => "N2"
```

Figure 4.7: Uncertainty Julia structure (writeUNC output)

Chapter 5

Solution Methodology

The problem solved by the planner is then a stochastic investment planning problem is formulated as:

$$\min_{x\in\mathcal{X}} f(x) + \sum_{i\in\mathcal{I}} \pi_i g\left(x_i, c_i\right),$$

where $\mathcal{I}$ is the set of decision nodes, each associated with a probability π_i.
The function:

$$g(x_i, c_i) = \min_{y_i\in\mathcal{Y}} \{c_i^{\mathrm{T}} C y_i | A y_i \leq B x_i\}, \quad \forall i \in \mathcal{I}$$

gives the cost of operating the system over the coming years.

In the planner, there are 3 different solution methdologies: Undecomposed, Standard Benders and Adaptive Benders. The undecomposed solution methodology involves only calling the solver on the complete model, rather than being a solution methodology developed for the planner, it is normally used for benchmarking purposes. The user can however benefit from the data manager module to assist in model building tasks.

The Standard Benders methodology, involves solving the operational model for every node in the planning tree, conversely, the Adaptive Benders methodology samples selected subproblems and shares information between these operational nodes to speed up the solution process.

The detailed information on the solution methodologies are outside of the scope of this manual but they will briefly mentioned at the end of this chapter.

5.1 Functions

5.1.1 generate_structures

The current version of the planner, is the evolution of the original planner (`https://github.com/nimazzi/Stand_and_Adapt_Bend`) written by Nicolò Mazzi. In the original code, all the solving functions where built on the information given by 2 structures, one called **B** which holds all the master problem data (including the optimization model) and **S** which holds the subproblem information.

- Location: */functions/load_functions.jl*
- Description: given the inputs required, it will return the B and S structures with the populated models and data, necessary to run the planner.
- Usage: *generate_structures(a,ms,mp,ps,pp,unc,mMP,mSP,i2n;mLP=Model(),mU=Model())* as shown takes several arguments. **a** is the algorithm structure, **ms**,**mp**,**ps** and **pp** are the sets and parameters for the master and subproblem, **mMP** is the object for the master problem model, **mSP** is the object of the subproblem model, **i2n** is the index-to-node dictioanry and the others **mLP** and **mU** are optional objects which correspond to the level (stabilisation) model and undecomposed model, respectively.

5.1.2 fillEmptyStructure

In Julia, structures and mutable structures can be easily defined by specifying each of the elements that will be considered in the structure and its data type. To initialize an object for the structure, the structure needs to be called and the arguments it receive must match the structure definition.

A simple example to understand this process is the following:

```julia
mutable struct example
    a::Int64
    b::Array{Float64,1}
end
```

Now, to initialize an object of type *example* a function call needs to be performed:

```julia
obj=example(2,[2.3,1.2])
```

This has created an object called **obj**, whose element *obj.a* will return 2 and *obj.b* will return [2.3, 1.2].
In large structures it may result cumbersome to have to modify the object generation when the structure is modifid, for instance, when new data wants to be collected; here the function *fillEmptyStructure* is a metaprogramming script, that will read the structure and depending on the fields and keys inside it, will automatically create an empty object.
An example of why this is convenient will be shown in the explicit example later in this guide.

- Location: */functions/load_support.jl*
- Description: generates an empty instance (object) of a structure type.
- Usage: *fillEmptyStructure(name::String)* as shown takes only as the argument, the name of the structure as a string. In this particular implementation, and based on the original planner, all the structures have a suffix "_type" so for instance.

5.1.3 setSP!

For Adaptive Benders, a "special point" is needed to always find a feasible point for the oracles. For more information see [1].

In this implementation of the method, the special point is set with the custom function *setSP!*. The information required to set it has already been specified and defined in the Linking Tables section, please refer to it, to see how to specify it.

- Location: */functions/load_support.jl*
- Description: sets in the models the special point, which will be used in the *step_0!* function if the solution methodology is Adaptive Benders.
- Usage: *setSP!(mpData,linkingTable* as shown takes 2 arguments, the master problem data contained in the **B** structure, and the linking table generated in previous steps.

5.1.4 solve_Benders!

- Location: */functions/load_functions.jl*
- Description: it starts the solving algorithm, based on the created models and algorithm options.
- Usage: *solve_Benders!(b,s,n,a)* as shown takes 4 arguments, **b** and **s** are the data structures for the master problem and subproblem, respectively and they contain the optimisation models. **n** is an auxiliary structure where data is gathered for debugging when doing algorithm development. **a** contains the algorithm parameters e.g. max. number of iterations, stabilisation parameters, convergence tolerance, etcetera.

Chapter 6

Julia scripts and function containers

6.1 /main.jl

In Figure 6.1 the main.jl file is shown. As a general description, the code first requires the path where the model and data are held, then imports the packages and other functions, and whether or not to read and write a jld2 file (binary data file).

After that, the data from the xlsx files is read and imported into a data structure called "data". The algorithm information is then read (which includes the selection of solution methodology) and stores it in a variable called "A".

If the decomposition is choses as the solution methodology, it then impotes the data structures required for decomposition and other functions. Generates the data structures "B", "S" and "N", and calls the decomposition algorithm with the function solve_Benders!.

6.2 functions/link_models.jl

In Figure 6.2 the link_models.jl file is shown. As a general description, the code creates the model objects for the subproblem, master problem, stabilisation problem and undecomposed problems (if required), and then populates them with the relevant information (variables, constraints and objective). Then the linking routines create the addtional variables and constraints connecting the subproblem with the master problem.

If the problem is to be solved as undecomposed, then the model is solved in this script.

6.3 functions/generate_structures.jl

```julia

## Creates different data gathering structures, with the information loaded from
    functions/load_structures.jl.
if A.algm==2
    N=fillEmptyStructure("N2");
elseif A.algm==1
    N=fillEmptyStructure("N1");
end

## Creates B and S structures needed for the planner (when doing Benders)
B,S= generate_structures(A,ms,mp,ps,pp,unc,m2,m,i2n;mLP=m3);
## Sets the special point.
setSP!(B,lT);
```

```julia
cd(dirname(@__FILE__)); flush(stdout); println("*/"*repeat("-",40)*"/*");

# Path to the optimisation models
modelpath="Nicolo";

# Specifies the relative path where data is held
path="$modelpath/dataXLSX";

# Loads the required packages: CSV, DataFrames, Gurobi, JuMP, etc.
include("functions/load_packages.jl");
# Loads the data support functions
include("functions/load_support.jl");
# Contains the models (SP, RMP, LMP, Undecomposed)
include("$modelpath/load_models.jl");

#### Data Loading & saving to JLD2 file/Restoring from a JLD2 file
# Name of the index file
mainFile="main.xlsx";
# Name of jld2 file to be saved or read
jldFile="data.jld2";
# Flag to ask it to update the data, and generate a new jld2 file
updateData=true;

fileExists=jldFile in cd(readdir,path);
if !fileExists || updateData
    flush(stdout); print("Generating data tables...    "); a=time();
    # Data loaded from XLSX files into dictionaries (new version, experimental)
    data=generateTablesfromXLSXNew(path,mainFile);
    println("done in $(round(time()-a,digits=1))s ")
else
    flush(stdout); print("Restoring data tables...     "); a=time();
    data=load("$path/$jldFile","data");
    println("done in $(round(time()-a,digits=1))s ")
end

############# ALGORITHM GLOBAL PARAMETERS DEFINITION
# Loads the tables for the general setup of the solution algorithm
algDict=loadParameters(data["Algorithm"],"parameters");
# Creates a structure for the algorithm parameters
A=transformStructure(algDict,"A");

############### DATA LOADING
# Custom combination of functions, according to requirements of model (IMPORTANT)
include("$path/data_preparation.jl")
# Creates optimisation models and link them for decomposition
include("functions/link_models.jl")

############### PLANNER PART
#If solution algorithm is not undecomposed
if A.algm!=0
    # Add stochastic planner's structures
    include("functions/load_structures.jl");
    # Add stochastic planner's functions
    include("functions/load_functions.jl");
    # Generates objects B and S, neccesary for the planner to work [with data
    loaded here] and set the special point for adaptive Benders
    include("functions/generate_structures.jl")
    # Has the functions to print out the results
    include("functions/load_printfunctions.jl");
    # Starts solving the problem
    solve_Benders!(B,S,N,A);
end
```

Figure 6.1: main.jl

```julia
## Creates an empty object to be solved by Gurobi with some solver attributes
m = Model(optimizer_with_attributes(()->Gurobi.Optimizer(gurobi_env),"OutputFlag"
    =>0,"Method"=>2))
### Populates the subproblem
SP!(m,ps,pp);

### Creates an empty object to be solved by Gurobi with some solver attributes
m2 = Model(optimizer_with_attributes(()->Gurobi.Optimizer(gurobi_env),"OutputFlag"
    =>0,"Method"=>1))
## If stabilisation is off, then it solves the RMP by Interior Point
A.stab==0 ? JuMP.set_optimizer_attribute(m2,"Method",2) : nothing;
## Populates the master problem
RMP!(m2,ms,mp);

### If stabilisation is ON, then creates another object to solve auxiilary
    stabilisation problems
if A.stab==1
    m3 = Model(optimizer_with_attributes(()->Gurobi.Optimizer(gurobi_env),"
    OutputFlag"=>0,"Method"=>2)
    LMP!(m3,ms,mp);
else
    m3= Model();
end

## If the data is not stored as datafile in the disk or the info needs to be
    updated:
if !fileExists || updateData
    ## Creates linking table
    lT,i2n=linkingTable(data["Linkage"]);
    jldopen("$path/$jldFile", "a+") do f f["lT"]=lT; f["i2n"]=i2n; end
else
    ##If the data exists and is up to date, then it restores the linking table
    lT,i2n=load.("$path/$jldFile",["lT","i2n"]);
end

## Creates node-to-index map, from the index-to-node map
n2i=Dict(i2n[i]=>i for i in keys(i2n));

print("Linking models...            ")
## Modifies the m and m2 objects with auxiliary linking variables and constraints
addLinking!(lT);
if A.stab==1 addLinkingLMP!(lT,"m3") end

## Writes the uncertainty structure
unc=writeUNC(lT,"u",n2i,i2n);

##  If the algorithm selected for solution is undecomposed, then it creates a
    model object, populates it and solves it.
if A.algm==0
    mU = Model(optimizer_with_attributes(()->Gurobi.Optimizer(gurobi_env),"
    OutputFlag"=>1,"Method"=>2)
    ## Populates the undecomposed model
    Undecomposed!(mU,ms,mp,lT);
    optimize!(mU);
end
```

Figure 6.2: link_models.jl

Chapter 7

Examples

The planner comes with 4 different models with some variations. They are developed in different data sets and help answering different questions. It shows that different models can be solved with the Adaptive Benders methodology using the same piece of software.

7.1 Summary of available test instances

The included models are:

1. Nicolo
2. Hongyu
3. North Sea
4. Ken/Nagisa

7.1.1 Nicolo

Path: Nicolo. Single region GB model with 12 technologies for investment.

In terms of investment, there are 3 stages and the investments at the current stage will be available in the next stage. In operation terms, it solves 8760-period problems with 4 slices, including storage and ramping constraints. There are up to 4 independient uncertainties: Uranium price, CO2 price, Demand scaling and CO2 level.

Depending on the number of desired uncertainties, it has stochastic trees spanning from single 3 nodes to up to 6564 nodes. It is based on the case presented with the paper [1].

An alternative version of this instance exist, where the investment does not necesarilly become available in the next stage i.e. can take more than 1 stage to become available. The alternative version can also have uncertainty in availability, e.g. investment in Nuclear can be available 1 stage later with a 89.9% probability, in 2 stages with a 10% probability and could never be realised with a 0.1% probability. In these models, investment that becomes available immediately is possible. (Path: Nicolo_Sanity)

The datasets are written in XLSX files and also in CSV files, in case an example with CSV files is needed. (Path: Nicolo)

7.1.2 Hongyu

Path: Hongyu. 5 region GB model with 12 technologies per region, and 7 transmission lines. In terms of investment, it is similar to Nicolo's version, with 3 stages and the same type of uncertainties. In operational terms, apart from the regions, it also includes lines connecting them, and heat stores / heat pumps.

The operational model includes (currently commented out) a linear approximation of a gas network. The model then is able to decide whether to invest in gas boilers or in heat pumps for instance.

For more information refer to Hongyu Zhangs's MSc dissertation. [Reference missing]

7.1.3 North Sea

Path: NorthSea. 12 region model including GB and Norway, and the interconnects in the North Sea. In terms of investment, planned up to the year 2050 in 6 (deterministic) stages. It includes numerous generators (hundreds?). In operational terms it has multiple slices (called short-term scenarios) with additional constraints. It includes the concept of Power Hub.

For more information refer to Hongyu Zhang's model. [Reference missing]

7.1.4 Ken/Nagisa

Path: Nagisa. Greenfield (also investment tree version) model of GB with emphasis on thermal modelling, in particular with investment in insulation.

7.2 Nicolo's Example detailed explanation

As explained earlier, the objective of the problem is to minimise the total investment and expected operation cost attained in a 3 stage problem. The information passed to the subproblems is both the x_i and the c_i coefficients.

The vector of right-hand side coefficients x_i is given by:

$$x_i = \left(\{x_{pi}^{acc}, \forall p \in \mathcal{P}\}, -v_i^D, v_i^E\right), \quad \forall i \in \mathcal{I}$$

here x_{pi}^{acc} is the accumulated capacity of technology p at node i. Parameters v_i^D and v_i^E are the relative level of energy demand and the yearly CO_2 emission limit, respectively. The vector of cost coefficients c_i is:

$$c_i = (c_i^{nucl}, c_i^{CO2}), \quad \forall i \in \mathcal{I}$$

where c_i^{nucl} is the uranium fuel price and c_i^{CO2} the CO_2 emission price. Finally, the function $f(x)$ yields the expected total investment and fixed cost, and it is computed as:

$$f(x) = \sum_{i \in \mathcal{I}} \pi_i \sum_{p \in \mathcal{P}} \left(c_{pi}^{inv} x_{pi}^{inst} + c_{pi}^{fix} x_{pi}^{acc}\right)$$

In the following we go go though the problem formulation. To simplify the notation we assume there is only a single slice in the operational model.

7.2.1 Notation

To formulate the model we use the following notation.

Sets

$\mathcal{T}$	`ps.T`	set of thermal generators
$\mathcal{R}$	`ps.R`	set of renewable generators
$\mathcal{B}$	`ps.B`	set of storage
$\mathcal{P}$	`ms.G`	set of thermal and renewable generators and storage $= \mathcal{T} \cup \mathcal{R} \cup \mathcal{B}$
$\mathcal{N}$	`ps.N`	set of nuclear power plants, which is a subset of $\mathcal{T}$
$\mathcal{D}$	`ms.D`	Set of demand
$\mathcal{I}$	`ms.I`	set of nodes with an operational problem
$\mathcal{I}_0$	`ms.I0`	set of nodes with investment decisions, which is a subset of $\mathcal{I}$
$\mathcal{H}$	`ps.H`	set of time periods where operational decisions are taken $= \{1, 2, \ldots, \lvert\mathcal{H}\rvert\}$

Parameters

x_{pi}^{init}	`mp.xh[p][i]`	generator capacity available by default; $p \in \mathcal{P}, i \in \mathcal{I}$
η_p^{gen}	`pp.eta[p]`	efficiency; $p \in \mathcal{T}$
η_p^{bat}	`pp.eta[p]`	storage charge efficiency; $p \in \mathcal{B}$
ρ_p	`pp.pb[p]`	storage charge rate limit; $p \in \mathcal{B}$
a_p^{CO2}	`pp.eg[p]`	CO2 factor; $p \in \mathcal{P}$

c_p^{op}	`pp.cvg[p]`	operational cost; $p \in \mathcal{P}$
c_{pi}^{inv}	`mp.ci[p][i]`	investment cost; $p \in \mathcal{P}, i \in \mathcal{I}$
c_p^{fuel}	`pp.cfg[p]`	fuel cost; $p \in \mathcal{T} \backslash \mathcal{N}$
c_i^{CO2}	`mp.cco2[i]`	CO2 cost; $i \in \mathcal{I}$
c_d^{shed}	`pp.cs[d]`	load shed cost; $d \in \mathcal{D}$
d_{dh}	`pp.pd[d][h]`	demand in the base case; $d \in \mathcal{D}, h \in \mathcal{H}$
Δ_h	`pp.ts[HS[h]]`	length of the operational time period; $h \in \mathcal{H}$
R_p	`pp.rg[p]`	ramp limit; $p \in \mathcal{P}$
u_{ph}	`pp.pr[p][h]`	renewable power plants utilisation rate; $p \in \mathcal{P}, h \in \mathcal{H}$

Variables

x_{pi}^{inst}	`pN[p,i]`	newly installed capacity; $p \in \mathcal{P}, i \in \mathcal{I}_0$
x_{pi}^{acc}	`pC[p,i]`	cumulative capacity; $p \in \mathcal{P}, i \in \mathcal{I}$
y_{pih}^{gen}	`yG[i,p,h]`	generation level; $p \in \mathcal{T} \cup \mathcal{R}, i \in \mathcal{I}, h \in \mathcal{H}$
y_{pih}^{charge}	`yI[i,p,h]`	amount of power charged; $p \in \mathcal{B}, i \in \mathcal{I}, h \in \mathcal{H}$
y_{pih}^{disch}	`yO[i,p,h]`	amount of power discharged; $p \in \mathcal{B}, i \in \mathcal{I}, h \in \mathcal{H}$
y_{pih}^{level}	`yL[i,p,h]`	energy level; $p \in \mathcal{B}, i \in \mathcal{I}, h \in \mathcal{H}$
y_{dih}^{shed}	`yS[i,d,h]`	load shedding; $d \in \mathcal{D}, i \in \mathcal{I}, h \in \mathcal{H}$

7.2.2 Investment Model

Other than the variable bounds there is only a single constraint in the investment model, which is to make variable x^{inst} and x^{acc} work as we expect:

$$x_{pi}^{\text{acc}} = x_{pi}^{\text{init}} + \sum_{j \in I_i^{\text{ance}}} x_{pj}^{\text{inst}}, \qquad \forall p \in \mathcal{P}, \forall i \in \mathcal{I}, \tag{7.1a}$$

where I_i^{ance} is the set of all the ancestor nodes of node i. The variable bounds are as follows.

$$0 \le x_{pi}^{\text{inst}},\ 0 \le x_{pi}^{\text{acc}} \le \overline{x}_{pi}^{\text{acc}}, \qquad \forall p \in \mathcal{P}, \forall i \in \mathcal{I}. \tag{7.2a}$$

The code corresponding to the mathematical description of the investment part of the model presented here is contained inside the *RMP!* function. The function can be seen in Figure 7.1.

7.2.3 Operational Model

All the variables in the operational model are nonnegative. $g(x_i, c_i)$ $(i \in \mathcal{I})$ is given by

$$\begin{aligned} g(x_i, c_i) = {} & c_i^{\text{nucl}} \sum_{p \in \mathcal{N}} \sum_{h \in \mathcal{H}} \frac{w_h}{\eta_p^{\text{gen}}} y_{pih}^{\text{gen}} + c_i^{\text{CO2}} \sum_{p \in \mathcal{T}} \sum_{h \in \mathcal{H}} \frac{a_p^{\text{CO2}} w_h}{\eta_p^{\text{gen}}} y_{pih}^{\text{gen}} \\ & + \sum_{h \in \mathcal{H}} \left(\sum_{p \in \mathcal{T} \backslash \mathcal{N}} \left(c_p^{\text{op}} + \frac{c_p^{\text{fuel}}}{\eta_p^{\text{gen}}} \right) y_{pih}^{\text{gen}} + \sum_{p \in \mathcal{N}} c_p^{\text{op}} y_{pih}^{\text{gen}} + \sum_{d \in \mathcal{D}} c_d^{\text{shed}} y_{dih}^{\text{shed}} \right) w_h. \end{aligned}$$

We consider the following constraints in the operational model.

- **load balance**

$$\sum_{p \in \mathcal{T} \cup \mathcal{R}} y_{pih}^{\text{gen}} + \sum_{p \in \mathcal{B}} (y_{pih}^{\text{disch}} - y_{pih}^{\text{charge}}) \ge \sum_{d \in \mathcal{D}} (v_i^D d_{dh} - y_{dih}^{\text{shed}}), \qquad \forall i \in \mathcal{I}, \forall h \in \mathcal{H}. \tag{7.3a}$$

- **CO2 limit**

$$\sum_{p \in \mathcal{T}} \sum_{h \in \mathcal{H}} \frac{a_p^{\text{CO2}} w_h}{\eta_p^{\text{gen}}} y_{pih}^{\text{gen}} \le v_i^E, \qquad \forall i \in \mathcal{I}. \tag{7.4a}$$

- **power output bounds**

$$0 \le y_{pih}^{\text{gen}} \le x_{pi}^{\text{acc}}, \qquad \forall p \in \mathcal{T}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}. \tag{7.5a}$$

```julia

function RMP!(m,ms,mp)
    # */ -- variables ---------------------------------------------- /* #
    @variable(m, pN[ms.G,ms.I0] >= 0) # newly installed capacity
    @variable(m, pC[ms.G,ms.I] >= 0) # cummulative capacity
    @variable(m, beta[ms.I] >= 0) # operational cost of node i

    # */ -- auxiliary variables ------------------------------------ /* #
    @variable(m,dL[ms.D,ms.I]>=0); # Demand scaling
    @variable(m, co2l[ms.I]) # CO2 level fixing variable
    @variable(m, cco[ms.I]) # CO2 fixing variable
    @variable(m, cur[ms.I]) # CO2 fixing variable

    # */ -- obj function ------------------------------------------- /* #
    ## compute investment-only cost
    @expression(m, f,sum(mp.prob[i0]*sum(mp.ci[g][i0]*pN[g,i0] for g in ms.G) for i0 in ms.I0) + sum(mp.kappa[i]*mp.prob[i]*sum(mp.cf[g]*pC[g,i] for g in ms.G) for i in ms.I))
    @expression(m,o,sum(oCost(mp,beta,i) for i in ms.I)); ## operational part of the expected cost
    @objective(m, Min, f+o) # investment plus operational cost (10^6£)

    # */ -- constraints -------------------------------------------- /* #
    ## compute cumulative capacity at node i
    @constraint(m, c02[g=ms.G,i=ms.I], pC[g,i] == mp.xh[g][i] + sum(pN[g,i0] for i0 in ms.map[i]))
    @constraint(m, c03[g=ms.G,i=ms.I], pC[g,i] >= .0 ) # minimum cumulative capacity
    @constraint(m, c04[g=ms.G,i=ms.I], pC[g,i] <= mp.xm[g] ) # maximum cumulative capacity

    #Fix the variable values (this is rewritten every time SP is solved)
    for i in ms.I
        fix(cco[i],mp.cco2[i];force=true)  # CO2 Cost (Exogenous)
        fix(cur[i],mp.cur[i];force=true)   # Uranium Cost (Exogenous)
        fix(co2l[i],mp.co2l[i];force=true) # CO2 level (Exogenous)
        for d in ms.D
            fix(dL[d,i],mp.dsc[d][i];force=true) # Demand level (Exogenous)
        end
    end

    return m
end
```

Figure 7.1: RMP! for the Nicolo case

- **renewable power generation**

$$y^{\text{gen}}_{pih} = u_{ph} x^{\text{acc}}_{pi}, \qquad \forall p \in \mathcal{R}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}. \tag{7.6a}$$

- **ramp rate bounds**

$$y^{\text{gen}}_{pih} - y^{\text{gen}}_{pih-1} \leq \Delta_h R_p x^{\text{acc}}_{pi}, \qquad \forall p \in \mathcal{T}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}, \tag{7.7a}$$

$$y^{\text{gen}}_{pih-1} - y^{\text{gen}}_{pih} \leq \Delta_h R_p x^{\text{acc}}_{pi}, \qquad \forall p \in \mathcal{T}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}, \tag{7.8a}$$

where by y^{gen}_{pi0} we denote $y^{\text{gen}}_{pi|\mathcal{H}|}$.

- **storage energy level**

$$y^{\text{level}}_{pih} - y^{\text{level}}_{pih-1} = \Delta_h(\eta^{\text{bat}}_p y^{\text{charge}}_{pih} - y^{\text{disch}}_{pih}), \qquad \forall p \in \mathcal{B}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}, \tag{7.9a}$$

where by y^{level}_{pi0} we denote $y^{\text{level}}_{pi|\mathcal{H}|}$.

- **storage energy level bounds**

$$0 \leq y^{\text{level}}_{pih} \leq x^{\text{acc}}_{pi}, \qquad \forall p \in \mathcal{B}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}. \tag{7.10a}$$

- **storage charge/discharge rate bounds**

$$y^{\text{charge}}_{pih} \leq \rho_p x^{\text{acc}}_{pi}, \qquad \forall p \in \mathcal{B}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}. \tag{7.11a}$$

$$y^{\text{disch}}_{pih} \leq \rho_p x^{\text{acc}}_{pi}, \qquad \forall p \in \mathcal{B}, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}. \tag{7.12a}$$

The code corresponding to the mathematical description of the operational part of the model presented here is contained inside the *SP!* function. The function can be seen in Figure 7.2.

7.3 Running an existing model

Running a model in the planner is straightforward, but before running it ideally the user should choose the instance to solve and the algorithm parameters. This can be achieved first by selecting the folder it is pointing in the "mail.jl" file (See Figure 6.1), by setting the variables **modelpath** and **path**. On the algorithmic side, the parameters are defined in the data as it was explained in Section 2.1.2 (Page 9).

Once the case and algorithm parameters have been selected. The process of executing an existing model straightforward. The simplest way just involves navigating to the root folder of the planner, calling Julia and giving the name of the run file ("main.jl" in the examples), i.e. `julia main.jl`. See Figure 7.3

A "better" alternative is to execute Julia, and then to "include" (call) the script. This way Julia will remain open, and it is easier to retrieve solutions as they can be directly queried instead of having to save them to a text file. See Figure 7.4.

In Figure 7.5 an example of the decomposition log is shown.

7.4 Retrieving results

Retrieving results depends primarily on the name of the variables created in the model and such, each model will need to query (or store) different variables.

The optimization results e.g. Optimal Investment in a node, operational cost of a node and operational generation, etcetera, are stored either in the Reduced master problem model or in the subproblem model (undecomposed model also if the solution methodology is undecomposed). As explained earlier,

```julia

function SP!(m::JuMP.Model,ps,pp)::JuMP.Model

    # */ -- Auxiliary info ---------------------------------------- /* #
    HS = H2S(ps.H,pp.st,pp.ln); # map of period (h) to slice
    sc = Dict(h => pp.yr*pp.wg[HS[h]]/(pp.ln[HS[h]]*pp.ts[HS[h]]) for h in ps.H); # Auxiliary dictionary (optional) that contains the scaling factors given a period h

    # */ -- variables ---------------------------------------------- /* #
    @variable(m, cco) # operational cost dependent on uncertain cost cᵢ
    @variable(m, cur) # operational cost dependent on uncertain cost cᵢ
    @variable(m, c0) # operational cost independent of uncertain costs c
    @variable(m, yG[ps.T,ps.H] >= 0) # generation level of conventional generation
    @variable(m, yI[ps.B,ps.H] >= 0) # charging power of storage generation
    @variable(m, yO[ps.B,ps.H] >= 0) # discharging power of storage generation
    @variable(m, yL[ps.B,ps.H] >= 0) # energy level of storage generation
    @variable(m, yS[ps.D,ps.H] >= 0) # load shedding

    #### Investment variables (fixed by the Master Problem)
    @variable(m,pub[ps.TB]>=0);
    @variable(m,co2l>=0);
    @variable(m,rsc[ps.R]>=0);
    @variable(m,sdm[ps.D]>=0);

    # */ -- obj function ------------------------------------------- /* #
    @objective(m, Min, c0 ) # generation cost (10⁶£)

    # */ -- constraints -------------------------------------------- /* #
    @expression(m, c0, exp10(-6)*sum((sum((pp.cvg[g]+pp.cfg[g]/pp.eta[g])*yG[g,h] for g in ps.T0) + pp.cvg[ps.N[1]]*yG[ps.N[1],h] + sum(pp.cs[d]*yS[d,h] for d in ps.D))*sc[h] for h in ps.H))  # compute operational cost independent of uncertain costs c
    @expression(m, cco, exp10(-6)*sum(sum(pp.eg[g]/pp.eta[g]*yG[g,h] for g in ps.T)*sc[h] for h in ps.H)) # compute cost dependent on CO₂ emission cost
    @expression(m, cur, exp10(-6)*sum(sum(1/pp.eta[ps.N[1]]*yG[ps.N[1],h])*sc[h] for h in ps.H)) # compute cost dependent on nuclear fuel cost

    @constraint(m, c04[b in ps.B, h in ps.H], yL[b,h] - yL[b,prev(h,pp.st,pp.ln)] == pp.ts[HS[h]]*(pp.eta[b]*yI[b,h] - yO[b,h])) # storage energy balance
    @constraint(m, c05[g in ps.T, h in ps.H], yG[g,h] - yG[g,prev(h,pp.st,pp.ln)] <= pp.ts[HS[h]]*pp.rg[g]*pub[g]) # upward ramping limitation on conventional generation
    @constraint(m, c06[g in ps.T, h in ps.H], yG[g,prev(h,pp.st,pp.ln)] - yG[g,h] <= pp.ts[HS[h]]*pp.rg[g]*pub[g]) # downward ramping limitation on conventional generation
    @constraint(m, c07[g in ps.T,h in ps.H], yG[g,h] <= pub[g]) # maximum capacity limitation on conventional generation
    @constraint(m, c08[b in ps.B,h in ps.H], yI[b,h] <= pp.pb[b]*pub[b]) # maximum capacity limitation on storage charging power
    @constraint(m, c09[b in ps.B,h in ps.H], yO[b,h] <= pp.pb[b]*pub[b]) # maximum capacity limitation on storage discharging power
    @constraint(m, c10[b in ps.B,h in ps.H], yL[b,h] <= pub[b]) # maximum capacity limitation on storage energy level
    @constraint(m, c11, sum(sum(yG[g,h]*pp.eg[g]/pp.eta[g] for g in ps.T)*sc[h] for h in ps.H) <= pp.lco*co2l) # CO₂ emission budget
    @constraint(m, c12[h in ps.H], sum(yG[g,h] for g in ps.T) + sum(yO[b,h]-yI[b,h] for b in ps.B) + sum(yS[d,h] for d in ps.D) + sum(pp.pr[r][h]*rsc[r] for r in ps.R) >= sum(sdm[d]*pp.pd[d][h] for d in ps.D)) # energy demand

    return m
end
```

Figure 7.2: SP! for the Nicolo case

```
user@computer % julia main.jl
*/---------------------------------------------/*
Loading packages...          completed in 13.6s
Loading support functions... completed in 0.3s
Generating data tables...    completed in 11.7s
Loading SP model...          completed in 8.5s
Loading RMP model...         completed in 1.5s
Loading LMP model...         completed in 0.0s
Creating linking table...    completed in 2.2s
Linking models...            completed in 0.8s
Optimising undecomposed model...

Gurobi Optimizer version 9.5.1 build v9.5.1rc2 (mac64[x86])
Thread count: 6 physical cores, 12 logical processors
Optimize a model with 543200 rows, 280398 columns and 1867714 nonzeros

                  Objective                  Residual
Iter        Primal           Dual          Primal    Dual      Compl      Time
   0   6.12705007e+09 -1.20301358e+08   5.34e+09 9.69e-01  3.26e+07     2s
  75   1.37981026e+05  1.37981026e+05   1.20e-06 3.55e-12  4.61e-11    22s

Barrier solved model in 75 iterations and 22.00 seconds (17.29 work units)
Optimal objective 1.37981026e+05
```

Figure 7.3: Direct run from terminal

the models are contained inside the Planner data structures.

All the information about the models then needs to be queried directly from the **B** structure (RMP) and **S** structure (SP).

The model information is stored in:

`B.rmp.m`	Object model for RMP
`S.ex.m`	Object model for SP
`mU`	Object undecomposed model

Depending on the type of data to be retrieved from the models, the user may need to different JuMP functions, such as *value*, *objective_value* or *dual.* Please refer to the standard JuMP documentation (`https://jump.dev/JuMP.jl/stable/`).

In Figure 7.6 shows retrieving from undecomposed problems, Figure 7.7 the process of querying investment information is shown. Figure 7.8 shows information for querying subproblem information and finally, .

In Figure 7.6, after running the optimization process, the user has retrieved the names of the variables contained in **pN**, note that the general way of retrieving information in a model is to pass the names as symbols (hence the colon symbol **:**). Then the user wants to see the optimal variable values for the newly invested capacity with the function value. Later the user requested the objective value of the problem, then the dual value on the scaling of the demand. If a specific index is required then it can be retrieved as shown in the last command.

To write the information to a text file (e.g. CSV), the user should use standard Julia commands. A quick overview of how to do it can be seen in `https://en.m.wikibooks.org/wiki/Introducing_Julia/Working_with_text_files`. Please refer to this source or to Julia's standard documentation.

Finally, the process to query, for example, the generation level in each period in the operational problem is shown in Figure 7.8.

```
bash$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.4.0 (2020-03-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("main.jl");

*/-----------------------------------------------/*
Loading packages...           completed in 13.6s
Loading support functions...  completed in 0.3s
Generating data tables...     completed in 11.7s
Loading SP model...           completed in 8.5s
Loading RMP model...          completed in 1.5s
Loading LMP model...          completed in 0.0s
Creating linking table...     completed in 2.2s
Linking models...             completed in 0.8s
Optimising undecomposed model...

Gurobi Optimizer version 9.5.1 build v9.5.1rc2 (mac64[x86])
Thread count: 6 physical cores, 12 logical processors
Optimize a model with 543200 rows, 280398 columns and 1867714 nonzeros

                  Objective                 Residual
Iter        Primal          Dual          Primal    Dual     Compl     Time
   0   6.12705007e+09 -1.20301358e+08  5.34e+09 9.69e-01  3.26e+07     2s
  75   1.37981026e+05  1.37981026e+05  1.20e-06 3.55e-12  4.61e-11    22s

Barrier solved model in 75 iterations and 22.00 seconds (17.29 work units)
Optimal objective 1.37981026e+05

julia>
```

Figure 7.4: Opening Julia and then running the script

```
julia> include("main.jl");

Loading packages...  done in 0.1s
Generating data tables...done in 20.7s
Loading SP model...  done in 2.1s
Loading RMP model... done in 0.6s
Creating linking table...done in 1.4s
Linking models...done in 0.3s
Load decomp. functions...done in 1.0s
Generating B,S structures... done in 4.2s
 */------------------------------------/*
 algorithm   : adapt_bend
 case    : 0
 subs per iter   : adaptive selection
 investment   nodes   : 2
 operational nodes   : 2
 */------------------------------------/*
 k =   1, n = 1,   = 99.7340 %
 k =   2, n = 1,   = 99.5090 %
 k =   3, n = 1,   = 99.4440 %
 k =   4, n = 1,   = 98.0560 %
 k =   5, n = 1,   = 95.6360 %
 k =   6, n = 1,   = 95.4000 %
 k =   7, n = 1,   = 92.2670 %
 k =   8, n = 1,   = 88.9250 %
 k =   9, n = 1,   = 87.6140 %
 k =  10, n = 2,   = 49.6320 %
 k =  11, n = 2,   = 37.1080 %
 k =  12, n = 2,   = 10.8190 %
 k =  13, n = 2,   =  6.3880 %
 k =  14, n = 2,   =  1.9180 %
 k =  15, n = 2,   =  1.5200 %
 k =  16, n = 1,   =  1.4760 %
 k =  17, n = 2,   =  1.4610 %
 k =  18, n = 2,   =  1.0160 %
 k =  19, n = 1,   =  0.8830 %
 k =  20, n = 1,   =  0.7260 %
 k =  21, n = 1,   =  0.6430 %
 k =  22, n = 1,   =  0.4720 %
 k =  23, n = 1,   =  0.4480 %
 k =  24, n = 1,   =  0.3570 %
 k =  25, n = 1,   =  0.0890 %

computational results:
 ----------------------------------------------------------------
  -target (%) | iters time (s) RMP!(%) SP(%)   Oracles(%)
 ----------------------------------------------------------------
 1.000        |    19     39      0.07   94.56   1.05
 0.100        |    25     48      0.08   96.06   0.75
 ----------------------------------------------------------------
LB: 137690.7530, UB: 137813.8299, total SP evaluations: 34
 */------------------------------------------------------------/*

julia>
```

Figure 7.5: Running Adaptive Benders log

```
julia> mU[:pN]
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, ["Gen1", "Gen2", "Gen3", ... , "Gen12"]
    Dimension 2, ["N1", "N2"]
And data, a 12×2 Matrix{VariableRef}:
 pN[Gen1,N1]    pN[Gen1,N2]
 pN[Gen2,N1]    pN[Gen2,N2]
 pN[Gen3,N1]    pN[Gen3,N2]
 pN[Gen4,N1]    pN[Gen4,N2]
 pN[Gen5,N1]    pN[Gen5,N2]
 pN[Gen6,N1]    pN[Gen6,N2]
 pN[Gen7,N1]    pN[Gen7,N2]
 pN[Gen8,N1]    pN[Gen8,N2]
 pN[Gen9,N1]    pN[Gen9,N2]
 pN[Gen10,N1]   pN[Gen10,N2]
 pN[Gen11,N1]   pN[Gen11,N2]
 pN[Gen12,N1]   pN[Gen12,N2]

julia> value.(mU[:pN])
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, ["Gen1", "Gen2", "Gen3", ... , "Gen12"]
    Dimension 2, ["N1", "N2"]
And data, a 12×2 Matrix{Float64}:
  0.0       0.0
  0.0       0.0
  0.0       0.0
 13.9437    2.3841
  1.15865   2.98895
  0.0       7.01115
  0.0       0.0
  0.0       0.0
  0.0       0.0
 19.0       0.0
  0.0       0.0
  0.0       0.0

julia> objective_value(mU)
137981.02623732222

julia> dual.(mU[:csr_dsc])
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, ["N2", "N3"]
    Dimension 2, ["D1"]
And data, a 2×1 Matrix{Float64}:
 72085.70997645342
 73931.15495731184

julia> value(mU[:pN]["Gen4","N1"])
 13.943650000000009

julia>
```

Figure 7.6: Querying data from the undecomposed problem

```
julia> value.(B.rmp.m[:pN])
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, ["Gen1", "Gen2", "Gen3", ... , "Gen12"]
    Dimension 2, ["N1", "N2"]
And data, a 12×2 Matrix{Float64}:
  0.0      0.0
  0.0      0.0
  0.0      0.0
 13.8186   2.92965
  1.27498  1.84199
  0.0      6.82827
  0.0      0.0
  0.0      0.0
  0.0      0.0
 19.0      0.0
  0.0      0.0
  0.0      0.0

julia>
```

Figure 7.7: Querying from Investment model

```
julia> value.(S.ex.m[:yG])
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, ["G1", "G2", "G3", "G4", "G5", "G6"]
    Dimension 2, 1:8760
And data, a 6×8760 Matrix{Float64}:
     0.0        0.0   …      0.0        0.0
  2000.0     2000.0       2000.0     2000.0
     0.0        0.0          0.0        0.0
  4722.24    3770.74      8415.29    8877.74
     0.0        0.0          0.0        0.0
 11882.0    11882.0   …  11882.0    11882.0

julia>
```

Figure 7.8: Querying info. from Operational model

Additionally, information of the algorithm can be queried in each iteration performed in the decomposition. This information may be relevant for algorithm development, but it is outside the scope of this section (it will be briefly mentioned in the algorithm development section).

7.5 Data Modification only example

To modify the data on the existing models it is only required to do it directly on CSV or XLSX files where the data is contained. For example in the case of Nicolo, if one generator has a different starting capacity (in the investment part), the data on the *dataG* sheet on the *Nicolo/dataXLSX/investment.xlsx* file can be modified, such that different nodes have different starting capacity.

To modify the operational data of a problem, the user needs to change the required values in the file *Nicolo/dataXLSX/operation.xlsx*, for example to change the ramp rates of different generators, it is enough to directly modify the values on the *gen_types* sheet with the desired ramping capacities.

7.6 Optimization Model modification example

In the Example presented in Section 7.2 a clear description of the model both in mathematical terms and in code has been presented. Modifying the optimization model means modifying the *SP!*, *RMP!*, *Undecomposed!* functions. If the model then requires additional data, the data structures need to be modified; and if the objective function is changing, the function *oCost* needs to be also modified.

A good example involves comparing Nicolo's models to Hongyu's models. Hongyu's models are significant modifications to data and models; they involve creating concepts of regions and heat-stores, and then the model uses these new elements to solve a 5-region model with heat.

7.7 Algorithm development

For the decompositions the steps followed are slightly different. The *solve_benders!* function calls the *step!* function, that executes different steps. What happens in each step is outlined in general here.

7.7.1 Standard Benders decomposition:

1. *step_a!*: the master problem is optimized, the lower bound and RMP decisions are stored.
2. *step_b!*: if stabilisation is activated, the stabilisation problem is solved and its decisions stored.
3. *step_c!*: RMP decisions are fixed in the subproblems, all subproblems are solved exactly, and its gradients and heights are communicated back to the RMP.
4. *step_e!*: the problem bounds and gaps are calculated
5. *step_f!*: cuts are added and used only in the subproblem in which they were generated.

7.7.2 Adaptive Benders decomposition:

1. *step_a!*: the master problem is optimized, the lower bound and RMP decisions are stored.
2. *step_b!*: if stabilisation is activated, the stabilisation problem is solved and its decisions stored.
3. *step_c!*: RMP decisions are fixed in the subproblems, the oracles are called to estimate upper and lower bounds in each subproblem, according to this information one or more suproblems are solved exactly; the exact solutions are stored in the oracles.
4. *step_e!*: the problem bounds and gaps are calculated based on the oracles.
5. *step_f!*: cuts are added with information from the oracles, so infromation is shared.

Addtionally, for adaptive Benders, there's an additional *step_0!* step that involves solving the special point as defined in the paper [1].

In reality, the code has several steps that have not been described in this section. Many of these steps were created to allow for different variations of the algorithms. Some other steps helps with stability and with speed. It is up to the developer to read the full source code.

Bibliography

[1] N. Mazzi, A. Grothey, K. McKinnon, and N. Sugishita, “Benders decomposition with adaptive oracles for large scale optimization,” *Mathematical Programming Computation*, vol. 13, pp. 683–703, Nov. 2020.

[2] A. Conejo, E. Castillo, R. Minguez, and R. Garcia-Bertrand, *Decomposition Techniques in Mathematical Programming: Engineering and Science Applications.* Springer, 2006.