MUSE

Release 0.9

Imperial College London

Nov 25, 2021

CONTENTS:

1 Glossary **1**

2 Installation **3**
- 2.1 For users . . . 4
- 2.2 For developers . . . 5

3 Running your first example **7**
- 3.1 Results . . . 7
- 3.2 Using Python . . . 8
- 3.3 Visualisation . . . 9
- 3.4 Next steps . . . 10

4 MUSE Overview **11**
- 4.1 How to use MUSE . . . 11
- 4.2 What questions can MUSE answer? . . . 12
- 4.3 What are MUSE's unique features? . . . 12
- 4.4 Visualisation of MUSE . . . 13
- 4.5 How MUSE works . . . 13
- 4.6 Foresight in MUSE . . . 14

5 Key MUSE Components **17**
- 5.1 Service Demand . . . 17
- 5.2 Technologies . . . 17
- 5.3 Sectors . . . 18
- 5.4 Agents . . . 18
- 5.5 Market Clearing Algorithm . . . 18

6 Customising MUSE Tutorials **19**
- 6.1 Adding a new technology . . . 19
- 6.2 Adding an agent . . . 27
- 6.3 Adding a region . . . 31
- 6.4 Modification of time . . . 34
- 6.5 Adding a service demand . . . 39
- 6.6 Adding a service demand by correlation . . . 45
- 6.7 Production constraints by timeslice . . . 48
- 6.8 Tutorial Information . . . 53

7 Input Files **55**
- 7.1 TOML primer . . . 55
- 7.2 Simulation settings . . . 56
- 7.3 Input Files . . . 64

7.4 Indices and tables . . . 76

8 Advanced guide **77**
8.1 Extending MUSE . . . 77
8.2 Further extending MUSE . . . 85
8.3 API . . . 90

9 FAQs **93**
9.1 I get a demand matching error . . . 93
9.2 What units should I be using within MUSE? . . . 94
9.3 How do I activate my conda environment? . . . 94
9.4 How do I know my conda environment is activated? . . . 94
9.5 I get a "Cannot find command 'git' - do you have 'git' installed in your PATH?" error . . . 94
9.6 When I input my GitHub password into Anaconda Powershell Prompt to download MUSE, I don't see any input . . . 94

10 API **95**
10.1 Market Clearing Algorithm . . . 96
10.2 Sectors and associated functionality . . . 96
10.3 Agents and associated functionalities . . . 96
10.4 Reading the inputs . . . 96
10.5 Writing Outputs . . . 96
10.6 Quantities . . . 96
10.7 Demand Matching Algorithm . . . 96
10.8 Miscellaneous . . . 96

11 Indices and tables **97**

Index **99**

CHAPTER

ONE

GLOSSARY

Here we provide a glossary for some of the frequently used terms in this documentation. Feel free to come back to this page when you come across a term that you don't understand!

Anaconda Anaconda is an open-source distribution of the Python and R programming languages. It is used for scientific computing. Anaconda comes with over 250 packages automatically installed, and allows for over 7,500 additional open-source packages to be installed via PyPI, the conda package and virrtual environment manager.

Anaconda Prompt Anaconda prompt is a command line shell. This is much like terminal for mac or cmd for windows. However, Anaconda Prompt allows you to run anaconda and conda commands from the prompt, without the need to change directories or your path.

Benchmark years These are the years which the model solves and presents outputs for. This can be set by the user and represents the evolution of the system in each time step. This, for example, can be set for every year or every five years, depending on the granularity required by the user.

Energy system An energy system is a system which is primarily designed to supply energy services to end-users. This can be defined as all components related to the production, conversion, delivery, and use of energy.

End-use demand The end-use demand is the demand for energy required to produce an energy service. For example, the gas required by a gas boiler to produce hot water, for the energy service of space heating.

Energy service The energy service is the service which a technology is used for. An example of an energy service could be cooking from a cooker.

End-use technology An end-use technology is a technology which service an energy service. Examples of these include boiler, heaters and stoves which service hot water, heating and cooking.

Endogenous quantity An endogenous quantity is one which develops within the model, as opposed to an exogenous quantity which is specified by the user and not influenced by the model.

Equilibrium An equilibrium is a situation in which economic forces, such as supply and demand are balanced. This means that, in the absence of external forces, the value of economic variables will not change.

Exogenous quantity An exogenous quantity is a quantity which is set by the user, and does not change based upon the simulation.

Imperfect information This is where an agent does not know everything needed to make a perfect decision.

Levelised cost of electricity The levelised cost of electricty is a measure of the average net present cost of electricity generation for a generating plant over its lifetime.

Limited foresight Limited foresight is the condition that an agent is unable to predict the entire future perfectly. The agent is only able to predict the future either imperfectly, or a limited time ahead.

Open-source Open-source is a source code that is made freely available for users to modify and redistribute.

Pandas Pandas is a data manipulation library for python.

Petajoules (PJ) A petajoule is equal to 1.0E+15 joules. A joule is a unit of energy and is equal to the energy transferred to an object.

Seaborn Seaborn is a visualisation library in python.

Scenario analysis Scenario analysis is a process of analysing the future by considering different possibilities of a future. Through this analysis multiple alternative future developments are presented, as opposed to a single prediction.

Simulation A simulation is computer software which models, or imitates a process or system. They can be used to observe what the effect of changes are over time to the system in question.

Timeslice A timeslice is the way in which a benchmark year is split into various different sections. For example, a benchmark year could be split into four seasons, or as far as for each hour within a benchmark year.

CHAPTER

TWO

INSTALLATION

There are two ways to install MUSE: one for users who do not wish to modify the source code of MUSE, and another for developers who do.

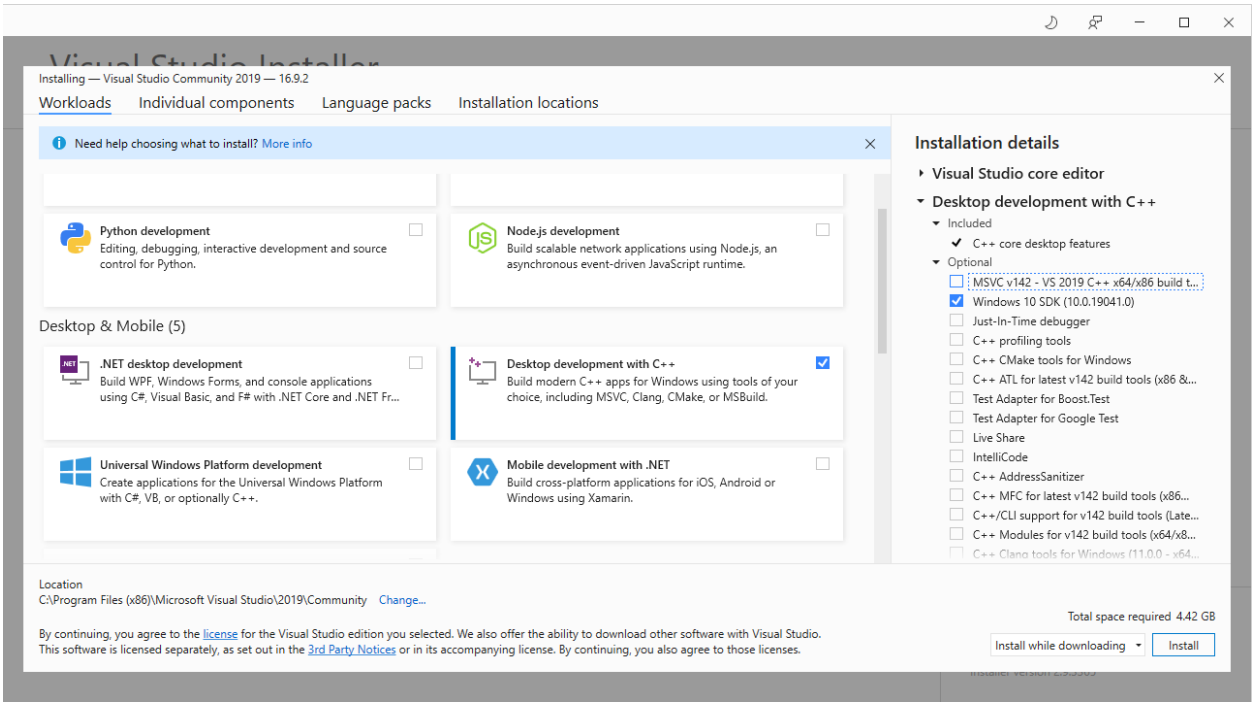
Note: Windows users and developers may need to install Windows Build Tools. These tools include C/C++ compilers which are needed to build some python dependencies.

MacOS includes compilers by default, hence no action is needed for Mac users.

Linux users may need to install a C compiler, whether GNU gcc or Clang, as well python development packages, depending on their distribution.

1. Install Visual Studio from the following link: https://visualstudio.microsoft.com/downloads/
2. Select your preferred edition. Although, the "Community" version is free and contains what is required.
3. Install Visual Studio by selecting the default options.
4. Download the Microsoft Visual C++ Build Tools from the following link by downloading Visual Studio 2019: https://visualstudio.microsoft.com/downloads/
5. Select your preferred edition. The "Community" is free and contains what is required.
6. Run the installer
7. Select: Workloads → Desktop development with C++.
8. Install options: select only the "Windows 10 SDK" (assuming the computer is Windows 10)]. This will come up on the right hand side of the screen.

The installation screen should look similar to the following:



For further information, see this link: https://www.scivision.dev/python-windows-visual-c-14-required

2.1 For users

MUSE is developed using python, an open-source programming language, which means that there are two steps to the installation process. First, python should be installed. Then so should MUSE.

The simplest method to install python is by downloading the Anaconda distribution. Make sure to choose the appropriate operating system (e.g. windows), python version 3.7, and the 64 bit installer. Once this has been done follow the steps for the anaconda installer, as prompted.

After python is installed we can install MUSE. MUSE can be installed via the Anaconda Prompt (or any terminal on Mac and Linux). This is a command-line interface to python and the python eco-system. In the anaconda prompt, run:

```
python -m pip install --user git+https://github.com/SGIModel/StarMuse
```

It should now be possible to run muse. Again, this can be done in the anaconda prompt as follows:

```
python -m muse --help
```

Note: Although not strictly necessary, users are encouraged to create an Anaconda virtual environment and install MUSE there, as shown in *For developers*.

2.2 For developers

Although not strictly necessary, creating an Anaconda virtual environment is highly recommended. Anaconda will isolate users and developers from changes occuring on their operating system, and from conflicts between python packages. It also ensures reproducibility from day to day.

Create a virtual env including python with:

```
conda create -n muse python=3.7
```

Activate the environment with:

```
conda activate muse
```

Later, to recover the system-wide "normal" python, deactivate the environment with:

```
conda deactivate
```

The simplest approach is to first download the muse code with git:

```
git clone https://github.com/SGIModel/StarMuse.git muse
```

For interested users, there are plenty of good tutorials for git. Next, it is possible to install the working directory into the conda environment:

```
# On Linux and Mac
cd muse
conda activate muse
python -m pip install -e ".[dev,docs]"

# On Windows
dir muse
conda activate muse
python -m pip install -e ".[dev,docs]"
```

The quotation marks are needed on some systems or shells, and do not hurt on any. The downloaded code can then be modified. The changes will be automatically reflected in the conda environment.

Tests can be run with the command pytest, from the testing framework of the same name.

The documentation can be built with:

```
python setup.py docs
```

The main page for the documentation can then be found at *build\sphinx\html\index.html* (or *build/sphinx/html/index.html* on Mac and Linux). The file can viewed from any web browser.

The source files to create the documentation can be found in the *docs/* folder from within the main MUSE directory.

CHAPTER

THREE

RUNNING YOUR FIRST EXAMPLE

In this section we run an example simulation of MUSE and visualise the results. There are a number of different examples in the source code, which can be found INSERT LINK HERE.

Once python and MUSE have been installed, we can run an example. To do this open anaconda prompt. Then change directory to where you have downloaded the MUSE source code.

Navigate to the following link for MacOS or Linux based operating systems:

```
{MUSE_download_location}/StarMuse/run/example/default/
```

Change `{MUSE_download_location}` to the location you downloaded MUSE to, for example `Users/{my_name}/Documents/` using the `cd` command, or "change directory" command. Once we have navigated to the directory containing the example settings `settings.toml` we can run the simulation using the following command in the anaconda prompt or terminal:

```
python -m muse settings.toml
```

If running correctly, your prompt should output text similar to that which can be found here.

It is also possible to run MUSE directly in python using the following code:

```
[ ]: from muse import examples
     model = examples.model("default")
     model.run()
```

3.1 Results

If the default MUSE example has run successfully, you should now have a folder called `Results` in the same directory as `settings.toml`.

This directory should contain results for each sector (`Gas`,`Power` and `Residential`) as well as results for the entire simulation in the form of `MCACapacity.csv` and `MCAPrices.csv`.

- `MCACapacity.csv` contains information about the capacity each agent has per technology per benchmark year. Each benchmark year is the modelled year in the `settings.toml` file. In our example, this is 2020, 2025, ..., 2050.
- `MCAPrices.csv` has the converged price of each commodity per benchmark year and timeslice. eg. the cost of electricity at night for electricity in 2020.

Within each of the sector result folders, there is an output for `Capacity` for each commodity in each year. The years into the future, which the simulation has not run to, refers to the capacity as it retires. Within the `Residential` folder there is also a folder for `Supply` within each year. This refers to how much end-use commodity was output.

The output can be fully configurable, as shown in the developer guide *here*.

3.2 Using Python

For the following parts of the tutorial, we will use python to run MUSE and visualise our results.

A common approach for data visualisation is to use Jupyter Notebook. Jupyter Notebook is a method of running interactive computing across dozens of programming languages. However, you are free to visualise the results using the language or program of your choice, for example Excel, R, Matlab or Python.

We will use Python, because MUSE is built using python, and so we can switch between running MUSE and visualisation rapidly.

First, you will need to install jupyter notebook in your anaconda environment. You will need to ensure that you have a conda environment activated. If you are using the Anaconda Powershell Prompt, you will see something similar to the following if your conda environment called muse is activated:

```
(muse) PS C:\Users\my_name>
```

If your environment is not activated you will see:

```
(base) PS C:\Users\my_name>
```

Once your environment is activated, you can installed Jupyter Notebook by following the instructions showed here. We will install the classic Jupyter Notebook, and so we will run the following code:

```
conda install -c conda-forge notebook
```

Once this has been installed you can start Jupyter Notebook by running the following command:

```
jupyter notebook
```

A web browser should now open up with a URL such as the following: `http://localhost:8888/tree`. If it doesn't, copy and paste the command as directed in the Anaconda Powershell Prompt. This will likely take the form of:

```
http://localhost:8888/?token=xxxxxxxxxx
```

Once you are on the page, you will be able to navigate to a location of your choice and create a new file, by clicking the `new` button in the top right, followed by clicking the `Python 3` button.

You should then be able to proceed and follow the tutorials in this documentation.

If you get any errors such as:

```
ModuleNotFoundError: No module named 'pandas'
```

It is possible to install the required packages by running the following in the Anaconda Powershell Prompt:

```
conda install pandas
```

or running the following in jupyter notebook:

```
!conda install pandas
```

3.3 Visualisation

[2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Next, we load the dataset of interest to us for this example: the `MCACapacity.csv` file. We do this using pandas.

[2]:
```
capacity_results = pd.read_csv("Results/MCAcapacity.csv")
capacity_results.head()
```

[2]:

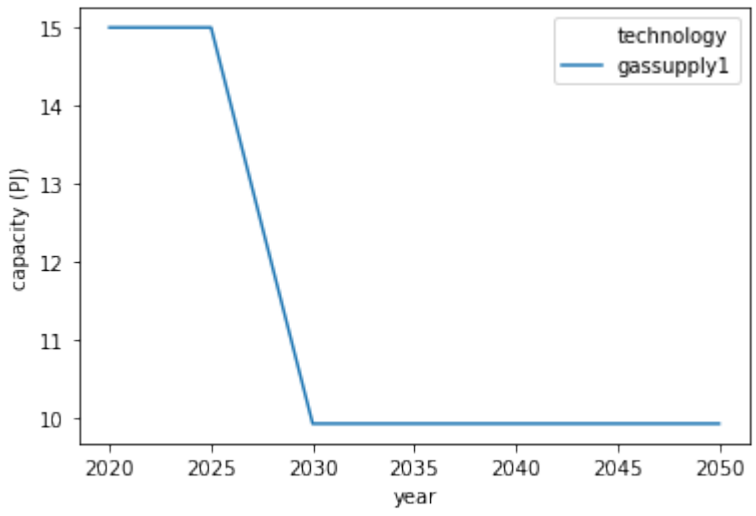
	technology	region	agent	type	sector	capacity	year
0	gasboiler	R1	A1	retrofit	residential	10.0	2020
1	gasCCGT	R1	A1	retrofit	power	1.0	2020
2	gassupply1	R1	A1	retrofit	gas	15.0	2020
3	gasboiler	R1	A1	retrofit	residential	5.0	2025
4	heatpump	R1	A1	retrofit	residential	19.0	2025

Using the `head` command we print the first five rows of our dataset. Next, we will visualise each of the sectors, with capacity on the y-axis and year on the x-axis.

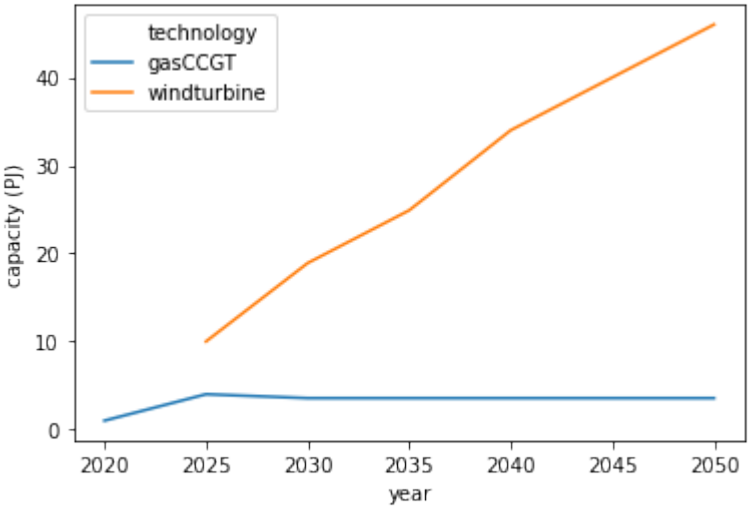
Don't worry too much about the code if some of it is unfamiliar. We effectively split the data into each sector and then plot a line plot for each.

[3]:
```
for sector_name, results in capacity_results.groupby("sector"):
    print("{} sector".format(sector_name))
    sns.lineplot(data=results, x="year", y="capacity", hue="technology")
    plt.ylabel("capacity (PJ)")
    plt.show()
    plt.close()
```

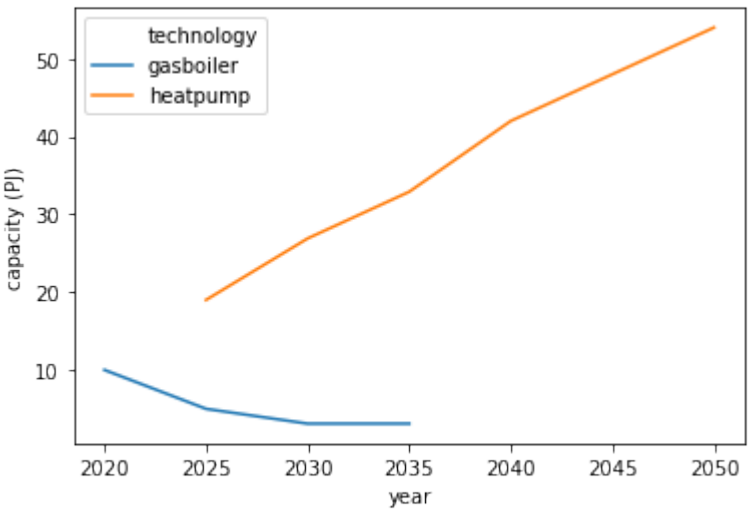
```
gas sector
```




```
power sector
```




```
residential sector
```



In this toy example, we can see that the end-use technology of choice in the residential sector becomes a heatpump. The heatpump displaces the gas boiler. Therefore, the supply of gas crashes due to a reduced demand. To account for the increase in demand for electricity, the agent invests heavily in wind turbines.

Note, that the units are in petajoules (PJ). MUSE requires consistent units across each of the sectors, and each of the input files (which we will see later). The model does not make any unit conversion internally.

3.4 Next steps

If you want to jump straight into customising your own example scenarios, head to the link *here*. If you would like a little bit of background based on how MUSE works first, head to the next section!

CHAPTER

FOUR

MUSE OVERVIEW

4.1 How to use MUSE

There are a huge number of ways that MUSE could be used. The energy field is varied and diverse, and many different scenarios can be explored. Users can model the impact of changes in technology prices, demand, policy instruments, sector interactions and much, much more. People are always thinking of new ways that MUSE can be used. So, get creative!

MUSE is an open source agent-based modelling environment that can be used to simulate change in an energy system over time. An example of the type of question MUSE can help in answering is:

- How may a carbon budget affect investments made in the power sector over the next 30 years?

MUSE can incorporate residential, power, industrial and conversion sectors, meaning many questions can be explored using MUSE, as per the wishes of the user.

MUSE is an agent-based modelling environment, where the agents are investors and consumers. In MUSE, this means that investment decisions are made from the point of view of the investor and consumer. These agents can be heterogenous, enabling for differering investment strategies between agents, as in the real world.

MUSE is technology rich and can model energy production, conversion and end-use technologies. So, for example, MUSE can enable the user to develop a power sector with solar photovoltaics, wind turbines and gas power plants which produce energy for appliances like electric stoves, heaters and lighting in the residential sector. Agents invest within these sectors, investing in technologies such as electric stoves in the residential sector or gas power plants in the power sectors. The investments made depend on the agent's investment strategies.

Every sector is a user configurable module. This means that a user can configure any number of sectors, cointaining custom, user-defined technologies and commodities. MUSE is fully data-driven, meaning that the configuration of the model is carried out using a selection of *Input Files*. This means that you are able to customise MUSE to your wishes by modifying these input files. Within a benchmark year, MUSE allows for a user-defined temporal granularity. This allows for the benchmark year to be split into different seasons and times, where energy demand may differ. Thus allowing us to model diurnal peaks in the demand, varying weekly and seasonally.

MUSE is highly configurable, but it has been built with medium and long-term scenarios in mind; for the short-term, MUSE can be linked with more detailed models. As the number of time steps and regions increase, the computational time also increases, which is something to keep in mind when building highly complex models.

Most energy systems models are based on cost optimisation and assume that all actors choose the cheapest available options. MUSE, however, uses a simulation framework which allows for the modelling of each sector according to the specific drivers triggering new investments in the sector. Despite the heterogeneity of these sectors, they interact with each other via the partial equilibrium approach which balances supply and demand of each commodity in the energy system.

As opposed to the majority of energy systems models, which assume that investors have full knowledge of future changes in the energy system across decades (intertemporal optimisation), MUSE uses a limited foresight approach. This allows

a user to define a configurable number of years over which the investors have knowledge of future commodity prices and demand. This more closely models the real-life case, we believe.

4.2 What questions can MUSE answer?

MUSE allows for users to investigate how an energy system may evolve over a time period, based upon investors using different decision metrics or objectives such as the net present value, levelized cost of electricity or a custom-defined function. In addition to this, it can simulate how investors search for technology options, and how different objectives are combined to reach an investment decision.

The search for new technologies can depend on several factors such as agents' budgets, technology maturity or preferences on the fuel-type. For instance, an investor in the power sector may decide that they want to focus on renewable energy, whereas another may prefer the perceived most profitable option.

Examples of the questions MUSE can answer include:

- How may India's steel industry decarbonise?
- How might residential consumers change their investment decisions over time?
- How might a carbon tax impact investments made in the power sector?

4.3 What are MUSE's unique features?

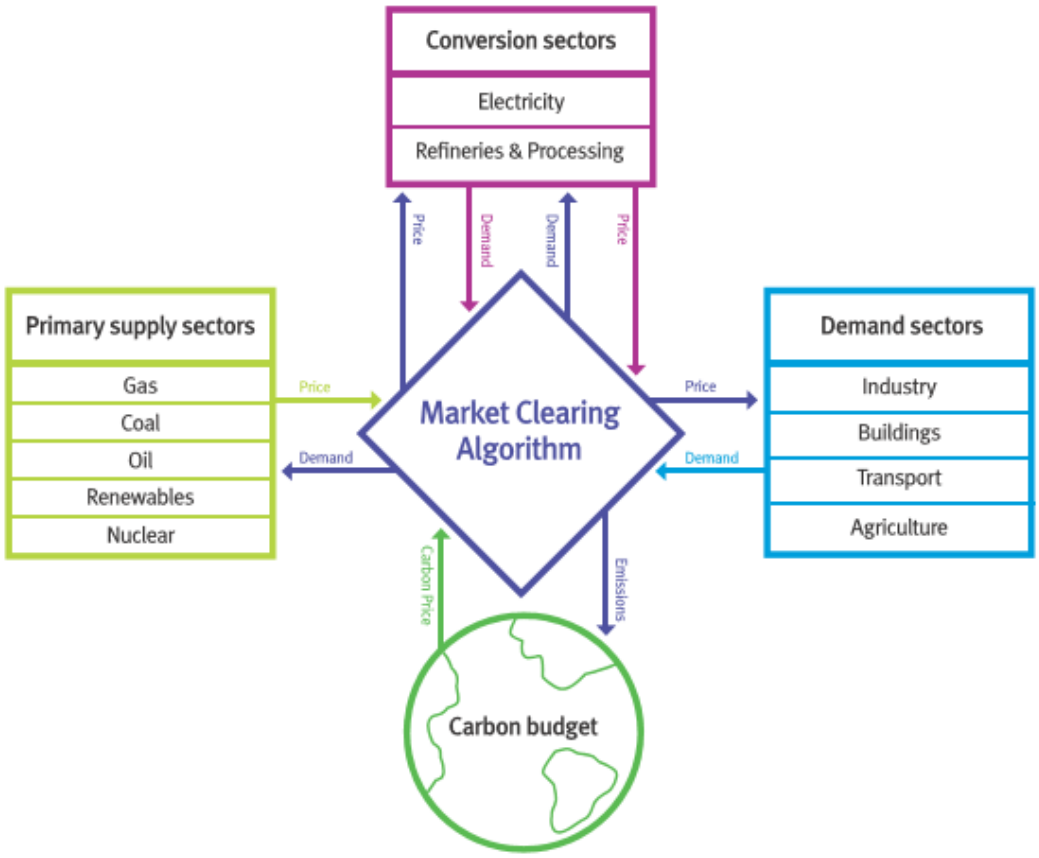
MUSE is a generalisable agent-based modelling environment and simulates energy transitions from the point of view of the investor and consumer agents. This means that users can define their own agents based upon their needs. The fact that MUSE is an agent-based model means that each of these agents can have different investment behaviours.

Additionally, agent-based models allow for agents to model imperfect information and limited foresight. An example of limited foresight is the ability to model the uncertainty residential users face when predicting the price of gas over the next 25 years. An example of imperfection information is that the technologies that agents invest in are tailored to represent the agents' attitude towards novel technologies and might not include all the technologies available. This is a unique feature to agent-based models when compared to intertemporal optimisation models and more closely models the real world. Many energy systems models are intertemporal optimisation models, which consider the viewpoint of a single benevolent decision maker, with perfect foresight and knowledge. These models optimise energy system investment and operation.

Whilst such intertemporal optimisation models are certainly useful, MUSE is different in that it models the incentives and challenges faced by investors. It can, therefore, be used to investigate different research questions, from the point of view of the investor and consumer. These questions are up to you, so impress us!

MUSE is completely open source, and ready for development.

4.4 Visualisation of MUSE



The figure above displays the key sectors of MUSE:

- Primary supply sectors; this allows to model diurnal peaks in the demand, varying weekly and seasonally.
- Conversion sectors
- Demand sectors
- Climate model (in the current model this is simplified by the use of a carbon budget.)
- Market clearing algorithm (MCA)

4.5 How MUSE works

MUSE works by iterating between sectors shown above to ensure that energy demands are met by the technologies chosen by the agents. Next, we detail the calculations made by MUSE throughout the simulation.

1. The service demand is calculated. For example, how much electricity, gas and oil demand is there for the energy services of cooking, building space heating and lighting in the residential sector? It must be noted, that this is only known after the energy service demand sector is solved and the technologies invested in are decided.
2. A demand sector is solved. That is, agents choose end-use technologies to serve the demands in the sector. For example, they compare electric stoves to gas stoves to meet the energy service demand of cooking. They then

choose between these technologies based upon their:

i. Search space (which technologies are they willing to consider?)

ii. Their objectives (which metrics do they consider important?)

iii. Their decision rules (how do they choose to combine their metrics if they have multiple?)

3. The decisions made by the agents in the demand sectors then leads to a certain level of demand for energy commodities, such as electricity, gas and oil, as a whole. This demand is then passed to the MCA.
4. The MCA then sends these demands to the sectors that supply these energy commodities (supply or conversion sectors).
5. The supply and conversion sectors are solved: agents in these sectors use the same approach (i.e. search space, objectives, decision rules) to decide which technologies to investment in to serve the energy commodity demand. For example, agents in the power sector may decide to invest in solar photovoltaics, wind turbines and gas power plants to service the electricity demand.
6. As a result of these decisions in supply and conversion sectors, a price for each energy commodity is formed. This price is formed based on the levelized cost of energy of the marginal technology. That, the technology which produces the marginal quantity. This price is then passed to the MCA.
7. The MCA then sends these prices back to the demand sectors, which are solved again as above.
8. This process repeats itself until commodity supply and demand converges for each energy commodity for each region. Once these converge, the model has found a "partial equilibrium" on the energy system and it moves forward to the next time period.

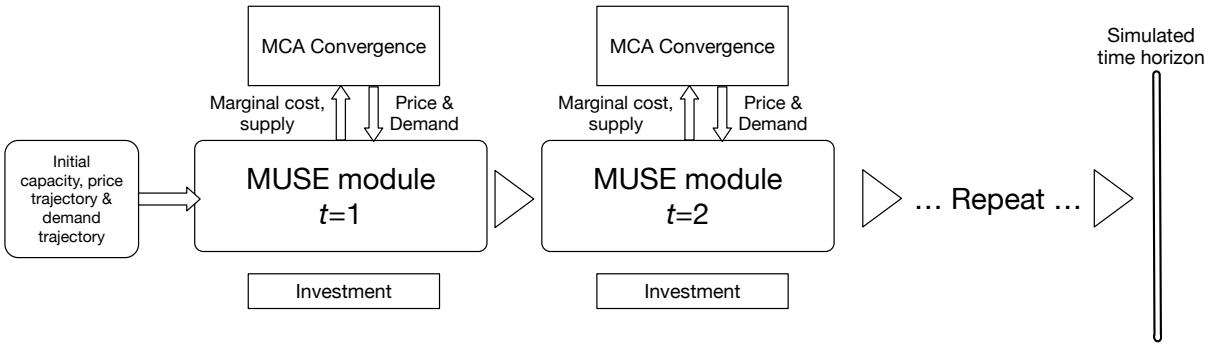
4.6 Foresight in MUSE

Within MUSE, investment decisions are made by the agents. To make these decisions, agents must use their limited knowledge of the future. This allows them to compare investment options under their expectations on prices and demand.

To model this process in MUSE, the agents are given limited foresight. The amount of limited foresight can be set by the user as a set of years. For example, if agents are given 5 years of limited foresight, they have certainty on the exogenous technology costs for the next 5 years. However, their expectations of future demand and prices for the lifetime of the plant, in that moment, are based on a flat-forward extension of the prices from the current period. However these prices can change in the next iteration. In contrast to perfect foresight, where variables such as prices, demand and technology costs in all the future time periods are known from the beginning of the simulation, using the limited foresight period, agents make investments under expectations of the market, which might be wrong.

The figure below details how MUSE runs. Firstly, the initial capacity, price trajectory and demand trajectory are known and set exogenously. These are used to initialize the MCA convergence algorithm. The MCA convergence algorithm finds a suitable set of investments which equilibrate supply and demand. Once equilibrium has been reached, technologies are decided and the commodity prices, which reflect the technology marginal costs. The investments balance asset retirements and the increase in demand, ensuring that supply meets demand.

This whole process repeats itself at every timestep until the specified number of milestone years have run.


MCA Convergence
MCA Convergence
Simulated time horizon
Marginal cost, supply
Price & Demand
Marginal cost, supply
Price & Demand
Initial capacity, price trajectory & demand trajectory
MUSE module t=1
MUSE module t=2
… Repeat …
Investment
Investment

CHAPTER

FIVE

KEY MUSE COMPONENTS

MUSE is made up of five key components:

- Service Demand
- Technologies
- Sectors
- Agents
- Market Clearing Algorithm

In this section we will briefly explore what these components do and and how they interact.

5.1 Service Demand

The energy service demand is a user input which defines the demand that an end-use sector has. An example of this is the service demand commodity of heat or cooling that the residential sector requires. End-use in this case, refers to the energy which is utilised at the very final stage, after both extraction and conversion.

The estimate of the energy service is the first step. This estimate can be an exogenous input derived from the user, or correlations of GDP and population which reflect the socio-economic development of a region or country.

5.2 Technologies

Users are able to define any technology they wish for each of the energy sectors. Examples include power generators such as coal power plants, buses in the transport sector or lighting in the residential sector.

Each of the technologies are placed in their regions of interest, such as the USA or India. They are then defined by the following, but not limited to, variables:

- Capital costs
- Fixed costs
- Maximum capacity limit
- Maximum capacity growth
- Lifetime of the technology
- Utilization factor
- Interest rate

Technologies, and their parameters are defined in the Technodata.csv file. For a full description of the input files, please refer to the *Techno-data Timeslices* file.

5.3 Sectors

Sectors typically group areas of economic activity together, such as the residential sector, which might include all energy conusming activies of households. Possible examples of sectors are:

- Gas sector
- Power sector
- Residential sector
- Industrial sector

Each of these sectors contain their respective technologies which consume energy commodities. For example, the residential sector may consume electricity, gas or oil for a variety of different energy demands such as lighting, cooking and heating.

Each of the technologies, which consume a commodity, also output a different commodity or service. For example, a gas boiler consumes gas, but outputs heat and hot water.

5.4 Agents

Agents represent the investment decision makers in an energy system, for example consumers or companies. They invest in technologies that meet service demands, like heating, or produce other needed energy commodities, like electricity. These agents can be heterogenous, meaning that their investment priorities have the ability to differ.

As an example, a generation company could compare potential power generators based on their levelized cost of electricity, their net present value, by minimising the total capital cost, a mixture of these and/or any user-defined approach. This approach more closely matches the behaviour of real-life agents in the energy market, where companies, or people, have different priorities and constraints.

5.5 Market Clearing Algorithm

The market clearing algorithm (MCA) is the central component between the different supplies and demands of the energy system in question. The MCA iterates between the demand and supply of each of these sectors. Its role is to govern the endogenous price of commodities over the course of a simulation.

For a hypothetical example, the price of electricity is set in a first iteration to $70/MWh. However, at this price, the majority of residential agents prefer to heat their homes using gas. As a result of this, residential agents consume less electricity and more gas. This reduction in demand reduces the electricity price to $50/MWh in the second iteration. However, at this lower electricity price, some agents decide to invest in electric heating as opposed to gas. Eventually, the price converges on $60/MWh, where supply and demand for both electricity and gas are equal.

This is the principle of the MCA. It finds an equilibrium by iterating through each of the different sectors until an overall equilibrium is reached for each of the commodities. It is possible to run the MCA in a carbon budget mode, as well as exogenous mode. The carbon budget mode ensures that an endogenous carbon price is calculated to limit the emissions of the energy system to be below a user-defined value. Whereas, the exogenous mode allows the carbon price to be set by the user.

CHAPTER

SIX

CUSTOMISING MUSE TUTORIALS

Next, we show you how to customise MUSE to create your own scenarios.

We recommend following the tutorials step by step, as the files build on the previous examples. If you prefer to jump straight in, your results may be different to the ones presented. To help you, we have provided the code to generate the various examples in case you want to compare your code to ours. Links to this code can be found in the table below, in the *Tutorial Information* section.

6.1 Adding a new technology

6.1.1 Input Files

MUSE is made up of a number of different *input files*. These, however, can be broadly split into two:

- *Simulation settings*
- *Simulation data*

Simulation settings specify how a simulation should be run. For example, which sectors to run, for how many years, the benchmark years and what to output. In this context, benchmark years are the years in which the model is solved. In the examples following, we solve for every 5 years, ie. 2020, 2025, 2030...

Whereas, simulation data parametrises the technologies involved in the simulation, or the number and kinds of agents.

To create a customised case study it is necessary to edit both of these file types.

Simulation settings are specified in a TOML file. TOML is a simple, extensible and intuitive file format well suited for specifying small sets of complex data.

Simulation data is specified in CSV. This is a common format used for larger datasets, and is made up of columns and rows, with a comma used to differentiate between entries.

MUSE requires at least the following files to successfully run:

- a single *simulation settings TOML file* for the simulation as a whole
- a file indicating initial market price *projections*
- a file describing the *commodities in the simulation*
- for generalized sectors:
- a file descring the *agents*
- a file descring the *technologies*
- a file descring the *input commodities* for each technology
- a file descring the *output commodities* for each technology

- a file descring the *existing capacity* of a given sector
- for each preset sector:
- a csv file describing consumption for the duration of the simulation

For a full description of these files see the *input files section*. To see how to customise an example, continue on this page.

6.1.2 Addition of solar PV

In this section, we will add solar photovoltaics to the default model seen in the *example page*. To achieve this, we must modify some of the input files shown in the above section. These files can be found in the `StarMuse` folder at the following location:

```
{muse_install_location}/src/muse/data/example/default
```

Change `{muse_install_location}` to the location where you installed MUSE using your file browser. You can modify the files in your favourite spreadsheet editor or text editor such as VSCODE, Excel, Numbers, Notepad or TextEdit.

6.1.3 Technodata Input

Within the default folder there is the `settings.toml` file, input folder and technodata folder. To add a technology within the power sector, we must open the `technodata` folder followed by the `power` folder.

At this point, we must note that we require consistency in input and output units. For example, if capacity is in PJ, the same basis would be needed for the output files `CommIn.csv` and `CommOut.csv`. In addition, across sectors a commodity needs to maintain the same unit. In these examples, we use the unit petajoule (PJ).

Next, we will edit the `CommIn.csv` file, which specifies the commodities consumed by solar photovoltaics.

The table below shows the original `CommIn.csv` version in normal text, and the added column and row in **bold**.

Process-Name	Region-Name	Time	Level	electricity	gas	heat	CO2f	wind	**solar**
Unit	•	Year	•	PJ/PJ	PJ/PJ	PJ/PJ	kt/PJ	PJ/PJ	**PJ/PJ**
gasCCGT	R1	2020	fixed	0	1.67	0	0	0	**0**
windturbine	R1	2020	fixed	0	0	0	0	1	**0**
solarPV	**R1**	**2020**	**fixed**	**0**	**0**	**0**	**0**	**0**	**1**

We must first add a new row at the bottom of the file, to indicate the new solar photovoltaic technology:

- we call this technology `solarPV`
- place it in region `R1`
- the data in this row is associated to the year 2020
- the input type is fixed
- solarPV consumes solar

As the solar commodity has not been previously defined, we must define it by adding a column, which we will call solar. We fill out the entries in the solar column, ie. that neither `gasCCGT` nor `windturbine` consume solar.

We repeat this process for the file: `CommOut.csv`. This file specifies the output of the technology. In our case, solar photovoltaics only output `electricity`. This is unlike `gasCCGT` which also outputs `CO2f`, or carbon dioxide.

Process-Name	Region-Name	Time	Level	electricity	gas	heat	CO2f	wind	**solar**
Unit	•	Year	•	PJ/PJ	PJ/PJ	PJ/PJ	kt/PJ	PJ/PJ	**PJ/PJ**
gasCCGT	R1	2020	fixed	1	0	0	91.67	0	**0**
windturbine	R1	2020	fixed	1	0	0	0	0	**0**
solarPV	**R1**	**2020**	**fixed**	**1**	**0**	**0**	**0**	**0**	**0**

Similar to the the `CommIn.csv`, we create a new row, and add in the solar commodity. We must ensure that we call our new commodity and technologies the same as the previous file for MUSE to successfully run. ie `solar` and `solarPV`.

Please note that we use flat forward extension of the values when only one value is defined. For example, in the `CommOut.csv` we only provide data for the year 2020. Therefore for the benchmark years, 2025, 2030, 2035… we assume the data remains unchanged from 2020.

The next file to modify is the `ExistingCapacity.csv` file. This file details the existing capacity of each technology, per benchmark year. For this example, we will set the existing capacity to be 0. Please note, that the model interpolates between years linearly.

ProcessName	RegionName	Unit	2020	2025	2030	2035	2040	2045	2050
gasCCGT	R1	PJ/y	1	1	0	0	0	0	0
windturbine	R1	PJ/y	0	0	0	0	0	0	0
solarPV	**R1**	**PJ/y**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Finally, the `technodata.csv` containts parametrisation data for the technology, such as the cost, growth constraints, lifetime of the power plant and fuel used. The technodata file is too long for it all to be displayed here, so we will truncate the full version.

Here, we will only define the parameters: `processName`, `RegionName`, `Time`, `Level`,`cap_par`, `Fuel`,`EndUse`,`Agent2` and `Agent1`

We shall copy the existing parameters from the `windturbine` technology for the remaining parameters that can be seen in the `technodata.csv` file for brevity. You can see the full file here INSERT LINK HERE.

Again, flat forward extension is used here. Therefore, as in this example we only provide data for the benchmark year 2020, 2025 and the following benchmark years will keep the same characteristics, e.g. costs, for each benchmark year of the simulation.

Process-Name	Region-Name	Time	Level	cap_par	cap_exp	…	Fuel	EndUse	Agent2
Unit	•	Year	•	MUS$2010/PJ_a	•	…	•	•	Retrofit
gasCCGT	R1	2020	fixed	23.78234399	1	…	gas	electricity	1
windturbine	R1	2020	fixed	36.30771182	1	…	wind	electricity	1
solarPV	**R1**	**2020**	**fixed**	**30**	**1**	…	**solar**	**electricity**	**1**

6.1.4 Global inputs

Next, navigate to the `input` folder, found at

```
{muse_installation_location}src/muse/data/example/default/input
```

We must now edit each of the files found here to add the new solar commodity. Due to space constraints we will not display all of the entries contained in the input files. The edited files can be viewed here INSERT LINK HERE however.

The `BaseYearExport.csv` file defines the exports in the base year. For our example we add a column to indicate that there is no export for solar. However, it is important that a column exists for our new commodity.

It is noted, however, that the `BaseYearImport.csv` as well as the `BaseYearExport.csv` files are optional files to define exogenous imports and exports; all values are set to zero if they are not used.

RegionName	Attribute	Time	electricity	gas	heat	CO2f	wind	**solar**
Unit	•	Year	PJ	PJ	PJ	kt	PJ	**PJ**
R1	Exports	2010	0	0	0	0	0	**0**
R1	Exports	2015	0	0	0	0	0	**0**
…	…	…	…	…	…	…	…	**…**
R1	Exports	2100	0	0	0	0	0	**0**

The `BaseYearImport.csv` file defines the imports in the base year. Similarly to `BaseYearExport.csv`, we add a column for solar in the `BaseYearImport.csv` file. Again, we indicate that solar has no imports.

RegionName	Attribute	Time	electricity	gas	heat	CO2f	wind	**solar**
Unit	•	Year	PJ	PJ	PJ	kt	PJ	**PJ**
R1	Imports	2010	0	0	0	0	0	**0**
R1	Imports	2015	0	0	0	0	0	**0**
…	…	…	…	…	…	…	…	**…**
R1	Imports	2100	0	0	0	0	0	**0**

The `GlobalCommodities.csv` file is the file which defines the commodities. Here we give the commodities a commodity type, CO2 emissions factor and heat rate. For this file, we will add the solar commodity, with zero CO2 emissions factor and a heat rate of 1.

Commodity	Commodity-Type	Commodity-Name	CommodityEmissionFactor_C O2	HeatRate	Unit
Electricity	Energy	electricity	0	1	PJ
Gas	Energy	gas	56.1	1	PJ
Heat	Energy	heat	0	1	PJ
Wind	Energy	wind	0	1	PJ
CO2fuelcomsbustion	Environmental	CO2f	0	1	kt
Solar	**Energy**	**solar**	**0**	**1**	**PJ**

The `projections.csv` file details the initial market prices for the commodities. The market clearing algorithm will update these throughout the simulation, however, an initial estimate is required to start the simulation. As solar energy is free, we will indicate this by adding a final column.

Please note that the unit row is not read by MUSE, but used as a reference for the user. The units should be consistent across all input files for MUSE; MUSE does not carry out any unit conversion.

RegionName	Attribute	Time	electricity	gas	heat	CO2f	wind	**solar**
Unit	•	Year	MUS$2010/PJ	MUS$2010/PJ	MUS$2010/PJ	MUS$2010/kt	MUS$2010/PJ	MUS$2010/PJ **
R1	CommodityPrice	2010	14.81481472	6.6759	100	0	0	**0**
R1	CommodityPrice	2015	17.89814806	6.914325	100	0.052913851	0	**0**
…	…	…	…	…	…	…	…	**…**
R1	CommodityPrice	2100	21.39814806	7.373485819	100	1.871299697	0	**0**

6.1.5 Running our customised simulation

Now we are able to run our simulation, with the new solar power technology.

To do this we run the same run command as previously in the anaconda command prompt:

```
python -m muse settings.toml
```

The output should be similar to the output here. However, expect the simulation to take slightly longer to run. This is due to the additional calculations made.

If the simulation has run successfully, you should now have a folder in the same location as your `settings.toml` file called `Results`. The next step is to visualise the results using the python visualisation package `seaborn` as well as the data analysis library `pandas`.

[1]:
```
import seaborn as sns
import pandas as pd
```

Next, we will import the `MCACapacity.csv` file into pandas and print the first 5 lines using the `head()` command.

Make sure to change the file path of `"../tutorial-code/1-add-new-technology/introduction/Results/MCACapacity.csv"` to where the `MCACapacity.csv` is on your computer, otherwise you will receive an error when you import the csv file.

[2]:
```
mca_capacity = pd.read_csv("../tutorial-code/1-add-new-technology/1-introduction/Results/
↪MCACapacity.csv")
mca_capacity.head()
```

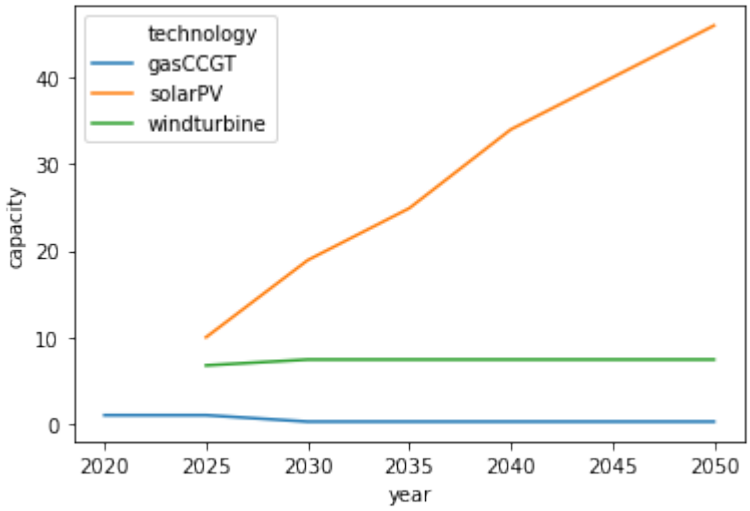
[2]:

	technology	region	agent	type	sector	capacity	year
0	gasboiler	R1	A1	retrofit	residential	10.0	2020
1	gasCCGT	R1	A1	retrofit	power	1.0	2020
2	gassupply1	R1	A1	retrofit	gas	15.0	2020
3	gasboiler	R1	A1	retrofit	residential	5.0	2025
4	heatpump	R1	A1	retrofit	residential	19.0	2025

We will only visualise the power sector in this example, as this was the only sector we changed. Therefore, we filter for this sector, and then visualise it using `seaborn`:

[3]:
```
power_capacity = mca_capacity[mca_capacity.sector=="power"]
sns.lineplot(data=power_capacity, x='year', y='capacity', hue="technology")
```

[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7faa52531e20>
```



We can now see that there is solarPV in addition to windturbine and gasCCGT, when compared to the example *here*! That's great and means it worked!

The difference in uptake of `solarPV` compared to `windturbine` is due to the fact that `solarPV` has a lower `cap_par` cost of 30, compared to the `windturbine`. Meaning that `solarPV` outcompetes both `windturbine` and `gasCCGT` in the electricity market.

6.1.6 Change Solar Price

Now, we will observe what happens if we increase the price of solar to be more expensive than wind in the year 2020, but then reduce the price of solar in 2040. By doing this, we should observe an increase in wind in the first few benchmark years of the simulation, followed by a transition to solar as we approach the year 2040.

To achieve this we have to modify the `Technodata.csv`, `CommIn.csv` and `CommOut.csv` files.

First, we will amend the `Technodata.csv` file as follows:

ProcessName	RegionName	Time	Level	cap_par	cap_exp	…	Fuel	EndUse	Agent2
Unit	•	Year	•	MUS$2010/PJ_a	•	…	•	•	Retrofit
gasCCGT	R1	2020	fixed	23.782343991	1	…	gas	electricity	1
gasCCGT	**R1**	**2040**	**fixed**	**23.782343991**	**1**	**…**	**gas**	**electricity**	**1**
windturbine	R1	2020	fixed	36.307711821	1	…	wind	electricity	1
windturbine	**R1**	**2040**	**fixed**	**36.307711821**	**1**	**…**	**wind**	**electricity**	**1**
solarPV	R1	2020	fixed	**40**	1	…	solar	electricity	1
solarPV	**R1**	**2040**	**fixed**	**30**	**1**	**…**	**solar**	**electricity**	**1**

Notice that we must provide entries for 2040 for the other technologies, `gasCCGT` and `windturbine`. For this example, we will keep these the same as before, copying and pasting the rows.

Here, we increase the `cap_par` variable by 10 for `solarPV` in the year 2020, to be a total of 40, and then reduce `cap_par` by 10 in 2040, again for `solarPV`.

MUSE uses interpolation for the years which are unknown. So in this example, for the benchmark years between 2020 and 2040 (2025, 2030, 2035), MUSE uses interpolated `cap_par` values. The interpolation mode can be set in the `settings.toml` file, and defaults to linear interpolation. This example uses the default setting for interpolation.

Next we will modify the `CommIn.csv` file.

For this step, we have to provide the input commodities for each technology, in each of the years defined in the `Technodata.csv` file. So, for this example we are required to provide entries for the years 2020 and 2040 for each of the technologies. For now, we won't change the 2040 values compared to the 2020. Therefore, we just need to copy and paste each of the entries for each of the technologies, as shown below:

Process-Name	Region-Name	Time	Level	electricity	gas	heat	CO2f	wind	solar
Unit	•	Year	•	PJ/PJ	PJ/PJ	PJ/PJ	kt/PJ	PJ/PJ	PJ/PJ
gasCCGT	R1	2020	fixed	0	1.67	0	0	0	0
gasCCGT	**R1**	**2040**	**fixed**	**0**	**1.67**	**0**	**0**	**0**	**0**
windturbine	R1	2020	fixed	0	0	0	0	1	0
windturbine	**R1**	**2040**	**fixed**	**0**	**0**	**0**	**0**	**1**	**0**
solarPV	R1	2020	fixed	0	0	0	0	0	1
solarPV	**R1**	**2040**	**fixed**	**0**	**0**	**0**	**0**	**0**	**1**

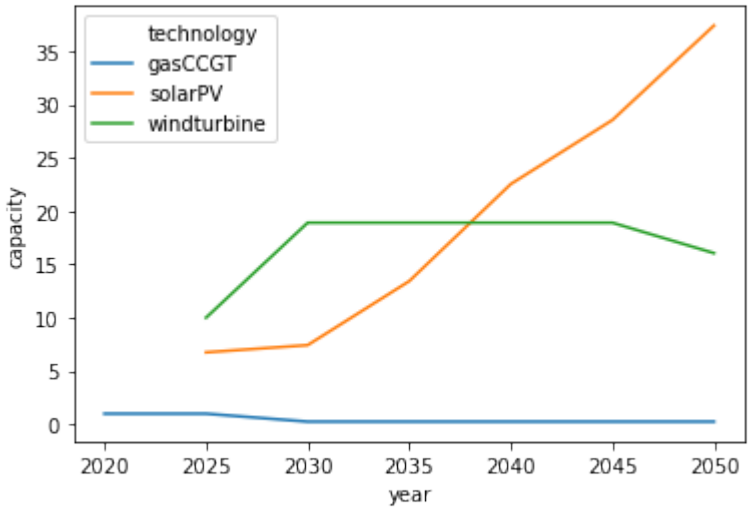
We must do the same for the `CommOut.csv` file. For the sake of brevity we won't show you this, but the link to the file can be found here INSERT LINK HERE.

We will now rerun the simulation, using the same command as previously and visualise the new results.

We must import the new `MCACapacity.csv` file again, and then visualise the results.

[4]:
```
mca_capacity = pd.read_csv("../tutorial-code/1-add-new-technology/2-scenario/Results/
↪MCACapacity.csv")
power_capacity = mca_capacity[mca_capacity.sector=="power"]
sns.lineplot(data=power_capacity, x='year', y='capacity', hue="technology")
```

[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7faa5261c0a0>
```



From the results, we can see that `windturbine` increases rapidly between 2025 and 2040. However, between the years 2030 and 2050 `solarPV` increases. This is because of the changing cost of `solarPV` during these years. A crossover can be seen around the year 2038.

For the full example with the completed input files see here INSERT LINK HERE

6.1.7 Next steps

In the next section we will add a new agent to the simulation. Note, that we will keep the additional `solarPV` technology, as well as the changing costs in 2040.

6.2 Adding an agent

In this section, we will add a new agent called `A2`. This agent will be slightly different to the other agents in the `default` example, in that it will make investments based upon a mixture of levelised cost of electricity (LCOE) and equivalent annual cost (EAC). These two objectives will be combined by calculating a weighted sum of the two when comparing potential investment options. We will give the LCOE a relative weight value of 1 and the EAC a relative weight value of 0.25.

We will continue to use the files from the previous tutorial where we added `solarPV` and created a new scenario.

To achieve this, first, we must modify the `Agents.csv` file in the directory:

```
{muse_install_location}/src/muse/data/example/default/technodata/Agents.csv
```

To do this, we will add two new rows to the file. To simplify the process, we copy the data from the first two rows of agent `A1`, changing only the rows: `AgentShare` `Name`, `Objective1`, `Objective2`, `ObjData1`, `ObjData2`, `DecisionMethod` and `Quantity`. The values we changed can be seen below. Notice how we edit the `AgentShare` column. This variable allows us to split the existing capacity between the two different agents. We will also need to edit the technodata file to define these new AgentShares.

Also notice that we amend the `Quantity` column. The reason for this is that we want to specify that Agent A1 makes up 50% of the population, and A2 makes up the remaining 50% of the population.

Again, we only show some of the rows due to space constraints, however see here INSERT LINK HERE for the full file.

AgentShare	Name	Region-Name	Objective1	Objective2	Objective3	ObjData1	ObjData2	…	Decision-Method	Quantity	…	Type
Agent1	A1	R1	LCOE			1		…	singleObj	0.5	…	New
Agent2	A1	R1	LCOE			1		…	singleObj	0.5	…	Retrofit
Agent3	**A2**	**R1**	**LCOE**	**EAC**		**0.5**	**0.5**	**…**	**weighted_sum**	0.5	**…**	**New**
Agent4	**A2**	**R1**	**LCOE**	**EAC**		**0.5**	**0.5**	**…**	**weighted_sum**	0.5	**…**	**Retrofit**

We then edit all of the `technodata` files to split the existing capacity between the two agents by the proportions we like. As we now have two agents which take up 50% of the population each, we will split the existing capacity by 50% for each of the agents. Notice that we only require the columns `Agent2` and `Agent4` to define the retrofit agents.

The new technodata file for the power sector will look like the following:

Process-Name	Region-Name	Time	Level	cap_par	cap_exp	...	Fuel	EndUse	Agent2	Agent4
Unit	•	Year	•	MUS$2010/PJ_a	•	...	•	•	Retrofit	**Retrofit**
gasCCGT	R1	2020	fixed	23.782343	1	...	gas	electricity	**0.5**	**0.5**
windturbine	R1	2020	fixed	36.307711	1	...	wind	electricity	**0.5**	**0.5**
solarPV	R1	2020	fixed	30	1	...	solar	electricity	**0.5**	**0.5**

However, remember you will have to make the same changes for the residential and gas sectors!

We will now save this file and run the new simulation model using the following command in Anaconda prompt:

```
python -m muse settings.toml
```

Again, we use seaborn and pandas to analyse the data in the `Results` folder.

[1]:
```
import pandas as pd
import seaborn as sns
```

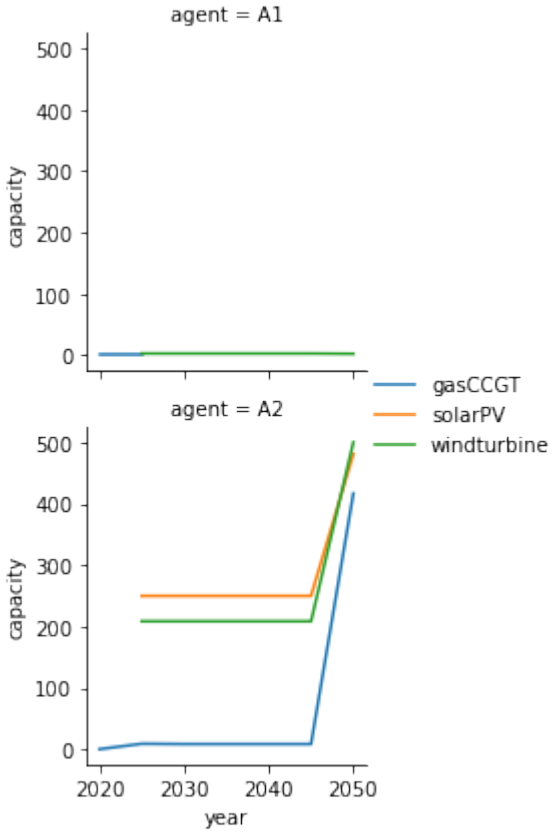
[2]:
```
mca_capacity = pd.read_csv("../tutorial-code/2-add-agent/1-multiple-objective/Results/
↪MCACapacity.csv")
power_sector = mca_capacity[mca_capacity.sector=="power"]
power_sector.head()
```

[2]:
```
    Unnamed: 0 agent  capacity dst_region  index  installed region sector  \
2            0    A1    0.5000         R1      0       2020     R1  power
3           21    A2    0.5000         R1      0       2020     R1  power
10           0    A1    0.5000         R1      0       2020     R1  power
11          14    A1    2.9549         R1     18       2020     R1  power
12          20    A2    9.3018         R1      0       2020     R1  power

     technology      type  year
2       gasCCGT  retrofit  2020
3       gasCCGT  retrofit  2020
10      gasCCGT  retrofit  2025
11  windturbine  retrofit  2025
12      gasCCGT  retrofit  2025
```

This time we can see that there is data for the new agent, `A2`. Next, we will visualise the investments made by each of the agents using seaborn's facetgrid command.

[3]:
```
power_sector = power_sector.groupby(["agent", "technology", "year"]).sum().reset_index()
g=sns.FacetGrid(power_sector, row='agent')
g.map(sns.lineplot, "year", "capacity", "technology")
g.add_legend()
```

[3]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe6893a0b80>
```



In this scenario we can see two divergent strategies. Agent `A2` invests heavily in `windturbine`, `gasCCGT` and `solarPV` early on. Whereas Agent `A1` invests a small amount in `windturbine` and `gasCCGT`.

From this small scenario, the difference between investment strategies between agents is evident. This is one of the key benefits of agent-based models when compared to optimisation based models.

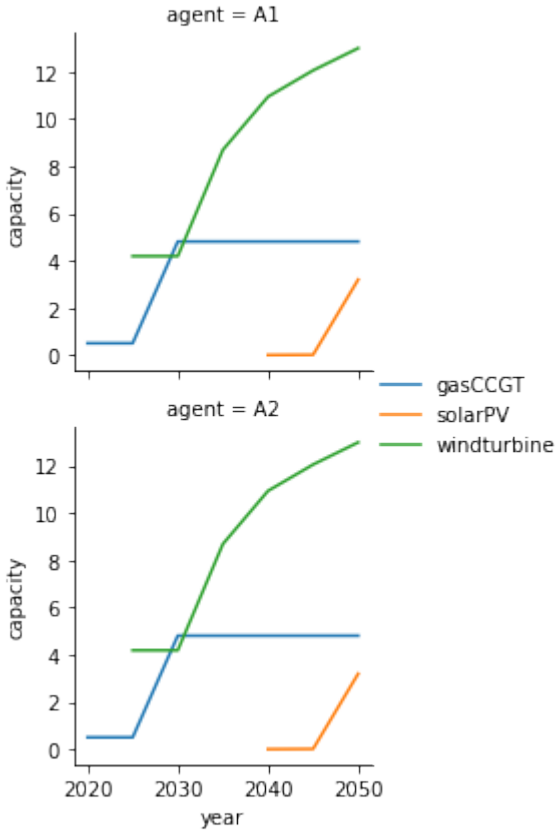
Next, we will see what occurs if the agents invest based upon the same investment strategy, with both investing using LCOE. This requires us to edit the `Agents.csv` file once more, to look like the following:

AgentShare	Name	Region-Name	Objective1	Objective2	Objective3	ObjData1	ObjData2	...	Decision-Method	...	Type
Agent1	A1	R1	LCOE			1		...	singleObj	...	New
Agent2	A1	R1	LCOE			1		...	singleObj	...	Retrofit
Agent3	A2	R1	**LCOE**			**1**		...	**singleObj**	...	New
Agent4	A2	R1	**LCOE**			**1**		...	**singleObj**	...	Retrofit

Again, this requires the re-running of the simulation, and to visualise the results like before:

[4]:
```
mca_capacity = pd.read_csv("../tutorial-code/2-add-agent/2-single-objective/Results/
↪MCACapacity.csv")
power_sector = mca_capacity[mca_capacity.sector=="power"]
power_sector = power_sector.groupby(["agent", "technology", "year"]).sum().reset_index()
g=sns.FacetGrid(power_sector, row='agent')
g.map(sns.lineplot, "year", "capacity", "technology")
g.add_legend()
```

[4]: <seaborn.axisgrid.FacetGrid at 0x7fe68628f100>



In this new scenario, with both agents running the same objective, very similar results between agents can be seen. There is a lower uptake in `gasCCGT`, and a large uptake in `windturbine`, whilst `solarPV` is invested in especially in the year 2040 where a change in the price of `solarPV` was set in the previous tutorial.

As the two agents make the same decisions, we see the same scenario develop for both agents, unlike in the previous scenario.

Have a play around with the files to see if you can come up with different scenarios!

6.2.1 Next steps

In the next section we will show you how to add a new region. We will maintain the two agents in this next section, and all the work done in the previous tutorials.

6.3 Adding a region

The next step is to add a region which we will call `R2`, however, this could equally be called `USA` or `India`. These regions do not have any energy trade. This requires us to undertake a similar process as before of modifying the input simulation data. However, this time we will also have to change the `settings.toml` file to achieve this.

The process to change the `settings.toml` file is relatively simple. We just have to add our new region to the `regions` variable, in the 4th line of the `settings.toml` file, like so:

```
regions = ["R1", "R2"]
```

The process to change the input files, however, takes a bit more time. To achieve this, there must be data for each of the sectors for the new region. This, therefore, requires the modification of every *input file*.

Due to space constraints, we will not show you how to edit all of the files. However, you can access the modified files here INSERT LINK HERE.

Effectively, for this example, we will copy and paste the results for each of the input files from region `R1`, and change the name of the region for the new rows to `R2`.

However, as we are increasing the demand by adding a region, as well as modifying the costs of technologies, it may be the case that a higher growth in technology is required. For example, there may be no possible solution to meet demand without increasing the `windturbine` maximum allowed limit. We will therefore increase the allowed limits for `windturbine` in region `R2`.

We have placed two examples as to how to edit the residential sector below. Again, the edited data are highlighted in **bold**, with the original data in normal text.

For the sake of brevity, we have omitted the entries for 2040 for the `CommIn.csv` file, however, just make sure to copy and paste the values for 2020 to 2040 here. The full file can be seen here INSERT LINK HERE.

The following file is the modified `/technodata/residential/CommIn.csv` file:

Process-Name	Region-Name	Time	Level	electric-ity	gas	heat	CO2f	wind
Unit	•	Year	•	PJ/PJ	PJ/PJ	PJ/PJ	kt/PJ	PJ/PJ
gasboiler	R1	2020	fixed	0	1.16	0	0	0
heatpump	R1	2020	fixed	0.4	0	0	0	0
gasboiler	**R2**	**2020**	**fixed**	**0**	**1.16**	**0**	**0**	**0**
heat-pump	**R2**	**2020**	**fixed**	**0.4**	**0**	**0**	**0**	**0**
…	…	…	…	…	…	…	…	…

Whereas the following file is the modified `/technodata/residential/ExistingCapacity.csv` file:

ProcessName	RegionName	Unit	2020	2025	2030	2035	2040	2045	2050
gasboiler	R1	PJ/y	10	5	0	0	0	0	0
heatpump	R1	PJ/y	0	0	0	0	0	0	0
gasboiler	**R2**	**PJ/y**	**10**	**5**	**0**	**0**	**0**	**0**	**0**
heatpump	**R2**	**PJ/y**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Below is the reduced `/technodata/power/technodata.csv` file, showing the increased capacity for `windturbine` in `R2`. For this, we highlight only the elements we changed from the rows in `R1`. The rest of the elements are the same for `R1` as they are for `R2`.

Again, we don't show the entries for 2040, apart from the edited windturbine row, for the sake of brevity.

ProcessName	RegionName	Time	...	MaxCapacityAddition	MaxCapacityGrowth	TotalCapacityLimit	...	Agent2
Unit	•	Year	...	PJ	%	PJ	...	Retrofit
gasCCGT	R1	2020	...	2	0.02	60	...	1
windturbine	R1	2020	...	2	0.02	60	...	1
solarPV	R1	2020	...	2	0.02	60	...	1
gasCCGT	R2	2020	...	2	0.02	60	...	1
windturbine	R2	2020	...	**5**	**0.05**	**100**	...	1
windturbine	R2	2040	...	**5**	**0.05**	**100**	...	1
solarPV	R2	2020	...	2	0.02	60	...	1
...	...	...	...	...	...	...	...	...

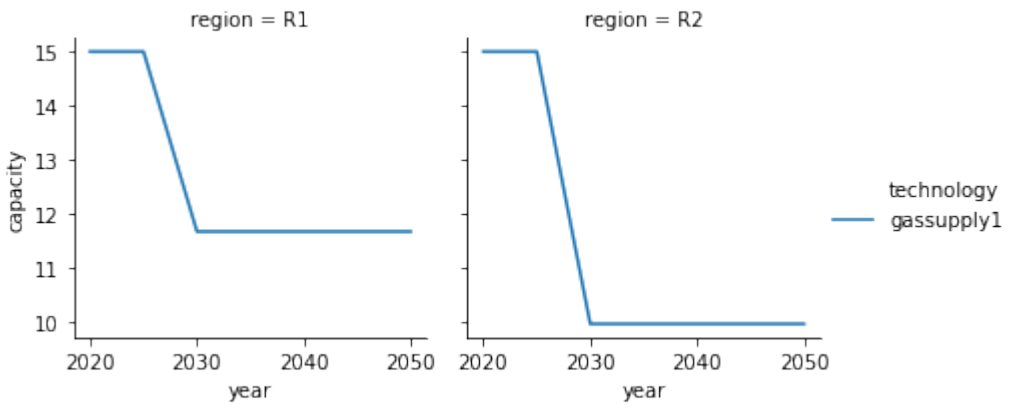
Now, go ahead and amend all of the other input files for each of the sectors by copying and pasting the rows from `R1` and replacing the `RegionName` to `R2` for the new rows. All of the edited input files can be seen here INSERT LINK HERE.

Again, we will run the results using the `python -m pip muse settings.toml` in anaconda prompt, and analyse the data as follows:

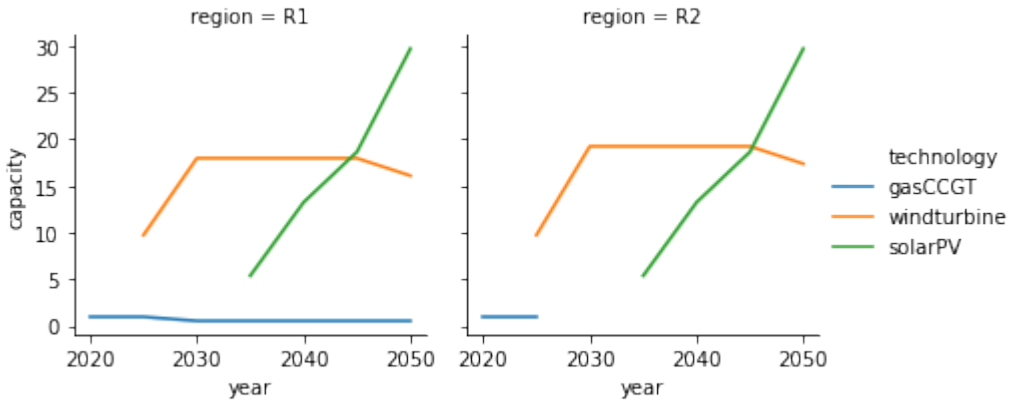
[1]:
```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
```

[2]:
```
mca_capacity = pd.read_csv("../tutorial-code/3-add-region/Results/MCACapacity.csv")

for name, sector in mca_capacity.groupby("sector"):
    print("{} sector:".format(name))
    g = sns.FacetGrid(data=sector, col="region")
    g.map(sns.lineplot, "year", "capacity", "technology")
    g.add_legend()
    plt.show()
```

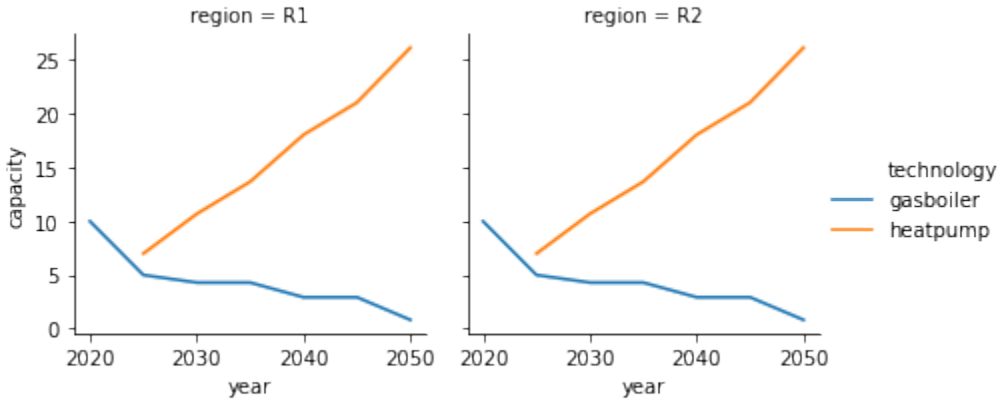
```
gas sector:
```



```
power sector:
```




```
residential sector:
```



Due to the similar natures of the two regions, with the parameters effectively copied and pasted between them, the results are very similar in both `R1` and `R2`. `gassupply1` drops significantly within the gas sector due to the carbon price profile. Which leads to the increasing demand of `heatpump` and falling demand of `gasboiler` in both region `R1` and `R2`. `windturbine` and `solarPV` increase significantly to match this demand of `heatpump`.

Have a play around with the various costs data in the technodata files for each of the sectors and technologies to see if different scenarios emerge. Although be careful. In some cases, the constraints on certain technologies will make it impossible for the demand to be met and it will give you an error such as the following:

```
message: 'The algorithm terminated successfully and determined that the problem is␣
↪infeasible.'
```

To avoid this error message you may have to relax the constraints in the technodata files. For instance, `MaxCapacityGrowth`.

6.3.1 Next steps

In the next section we modify the `settings.toml` file to change the timeslicing arrangements as well as project until 2040, instead of 2050, in two benchmark year time steps.

6.4 Modification of time

In this section we will show you how to modify the timeslicing arrangement as well as change the time horizon and benchmark year intervals by modifying the `settings.toml` file.

6.4.1 Modify timeslicing

Timeslicing is the division of a single benchmark year into multiple different sections. For example, we could slice the benchmark year into different seasons, make a distinction between weekday and weekend or a distinction between morning and night. We do this as energy demand profiles can show a difference between these timeslices. eg. Electricity consumption is lower during the night than during the day.

To achieve this, we have to modify the `settings.toml` file, as well as the files within the preset folder: `Residential2020Consumption.csv` and `Residential2050Consumption.csv`. This is so that we can edit the demand for the residential sector for the new timeslices.

First we edit the `settings.toml` file to add two additional timeslices: early-morning and late-afternoon. We also rename afternoon to mid-afternoon. These settings can be found at the bottom of the `settings.toml` file.

An example of the changes is shown below:

```
[timeslices]
all-year.all-week.night = 1095
all-year.all-week.morning = 1095
all-year.all-week.mid-afternoon = 1095
all-year.all-week.early-peak = 1095
all-year.all-week.late-peak = 1095
all-year.all-week.evening = 1095
all-year.all-week.early-morning = 1095
all-year.all-week.late-afternoon = 1095
level_names = ["month", "day", "hour"]
```

The number of timeslices within this should add up to 8760; the number of hours in a benchmark year. Whilst this is required, MUSE does not check and enforce this.

Next, we modify both Residential Consumption files. Again, we put the text in bold for the modified entries. We must add the demand for the two additional timelsices, which we call timeslice 7 and 8. We make the demand for heat to be 2 for both of the new timeslices.

Below is the modified `Residential2020Consumption.csv` file:

	RegionName	ProcessName	Timeslice	electricity	gas	heat	CO2f	wind
0	R1	gasboiler	1	0	0	1	0	0
1	R1	gasboiler	2	0	0	1.5	0	0
2	R1	gasboiler	3	0	0	1	0	0
3	R1	gasboiler	4	0	0	1.5	0	0
4	R1	gasboiler	5	0	0	3	0	0
5	R1	gasboiler	6	0	0	2	0	0
6	**R1**	**gasboiler**	**7**	**0**	**0**	**2**	**0**	**0**
7	**R1**	**gasboiler**	**8**	**0**	**0**	**2**	**0**	**0**
0	R2	gasboiler	1	0	0	1	0	0
1	R2	gasboiler	2	0	0	1.5	0	0
2	R2	gasboiler	3	0	0	1	0	0
3	R2	gasboiler	4	0	0	1.5	0	0
4	R2	gasboiler	5	0	0	3	0	0
5	R2	gasboiler	6	0	0	2	0	0
6	**R2**	**gasboiler**	**7**	**0**	**0**	**2**	**0**	**0**
7	**R2**	**gasboiler**	**8**	**0**	**0**	**2**	**0**	**0**

The `ProcessName` must be reported, but it is not binding on the results. It is just the way that the model reads the input data.

We do the same for the `Residential2050Consumption.csv`, however this time we make the demand for heat in 2050 to both be 5 for the new timeslices. See here INSERT LINK HERE for the full file.

Once the relevant files have been edited, we are able to run the simulation model using `python -m muse settings.toml`.

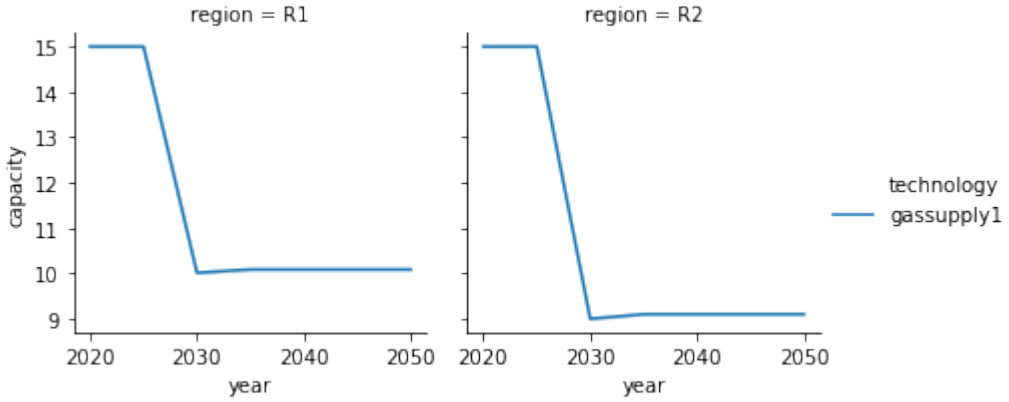
Then, once run, we import the necessary packages:

[1]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

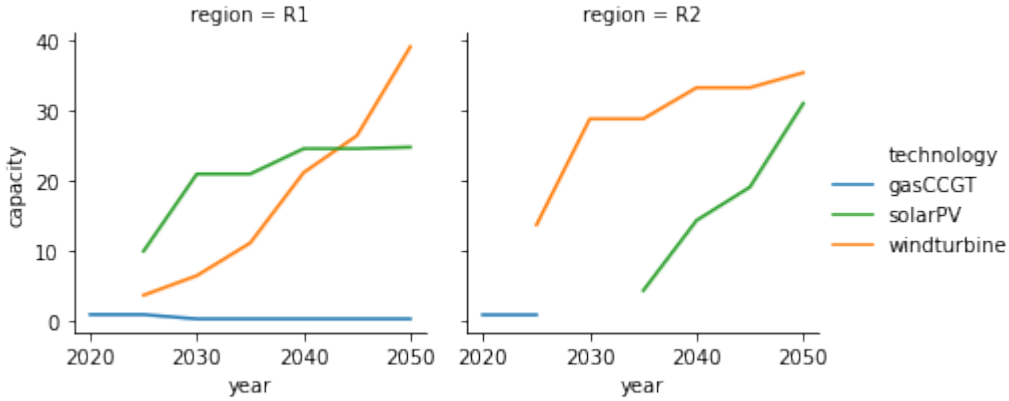
and visualise the relevant data:

[2]:
```
mca_capacity = pd.read_csv("../tutorial-code/4-modify-timing-data/1-modify-timeslices/
↪Results/MCACapacity.csv")

for name, sector in mca_capacity.groupby("sector"):
    print("{} sector:".format(name))
    g = sns.FacetGrid(data=sector, col="region")
    g.map(sns.lineplot, "year", "capacity", "technology")
    g.add_legend()
    plt.show()
    plt.close()
```

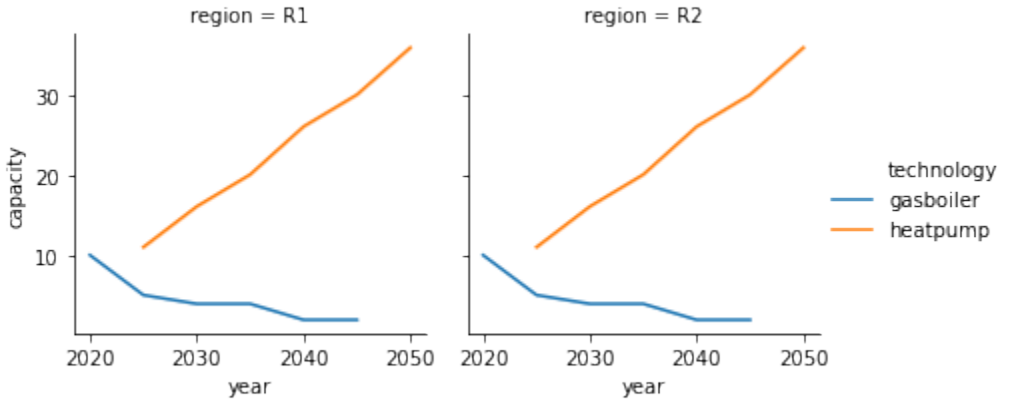
```
gas sector:
```




```
power sector:
```




```
residential sector:
```



Compared to the scenario where we added a *region*, there is a smaller increase in solarPV in region R2 in the power sector. However, the rest remains largely unchanged.

This example shows the trade-off between time granularity and speed of computation. This is due to the fact that as we add more timesteps, the model takes longer to run, but slightly different scenarios emerge. It is up to you to decide what level of granularity is required for your use case.

6.4.2 Modify time horizon and time periods

For the previous examples, we have run the scenario from 2020 to 2050, in 5 year time steps per benchmark year. This has been set at the top of the `settings.toml` file. However, we may want to run a more detailed scenario, with 2 year time steps, and up until the year 2040.

Making this change is quite simple as we only have two lines to change. We will modify line 2 and 3 of the `settings.toml` file, as follows:

```
# Global settings - most REQUIRED
time_framework = [2020, 2022, 2024, 2026, 2028, 2030, 2032, 2034, 2036, 2038, 2040]
foresight = 2   # Has to be a multiple of the minimum separation between the benchmark␣
↪years
```

The `time_framework` details each benchmark year in which we run the simulation. The `foresight` variable details how much foresight an agent has when making investments.

As we have modified the timeslicing arrangements there will be a change in the underlying demand for heating. This may require more electricity to service this demand. Therefore, we relax the constraints for growth in the power sector for all technologies and constraints in the `technodata/power/technodata.csv`, as shown below:

ProcessName	RegionName	…	MaxCapacityAddition	MaxCapacityGrowth	TotalCapacityLimit	…	Agent1
Unit	•	…	PJ	%	PJ	…	New
gasCCGT	R1	…	**40**	**0.2**	**120**	…	0
windturbine	R1	…	**40**	**0.2**	**120**	…	0
solarPV	R1	…	**40**	**0.2**	**120**	…	0
gasCCGT	R2	…	**40**	**0.2**	**120**	…	0
windturbine	R2	…	**40**	**0.2**	**120**	…	0
solarPV	R2	…	**40**	**0.2**	**120**	…	0
…	…	…	…	…	…	…	…

We also modify the constraints defined in the `technodata.csv` file for the residential sector, as shown below.

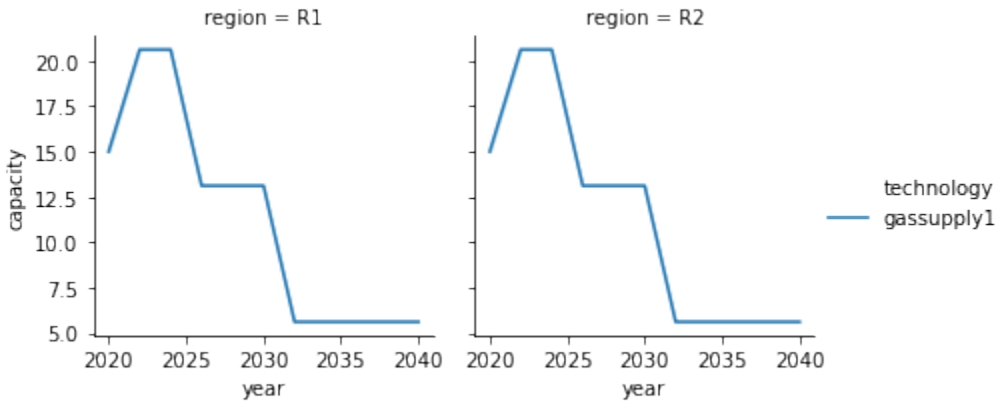
ProcessName	RegionName	Time	…	MaxCapacityAddition	MaxCapacityGrowth	TotalCapacityLimit	…	Agent1
Unit	•	Year	…	PJ	%	PJ	…	New
gasboiler	R1	2020	…	**60**	**0.5**	**120**	…	0
heatpump	R1	2020	…	**60**	**0.5**	**120**	…	0
gasboiler	R2	2020	…	**60**	**0.5**	**120**	…	0
heatpump	R2	2020	…	**60**	**0.5**	**120**	…	0

It must be noted, that this is a toy example. For modelling a real life scenario, data should be sought to ensure that these constriants remain realistic.

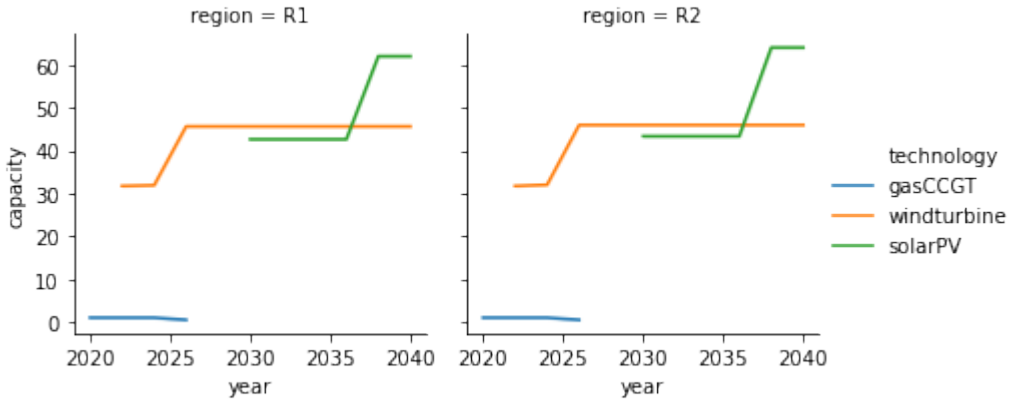
For the full power sector `technodata.csv` file click here INSERT LINK HERE, and for the full residential sector `technodata.csv` file click here INSERT LINK HERE.

[3]:
```
mca_capacity = pd.read_csv("../tutorial-code/4-modify-timing-data/2-modify-time-
↪framework/Results/MCACapacity.csv")

for name, sector in mca_capacity.groupby("sector"):
    print("{} sector:".format(name))
    g = sns.FacetGrid(data=sector, col="region")
    g.map(sns.lineplot, "year", "capacity", "technology")
    g.add_legend()
    plt.show()
    plt.close()
```

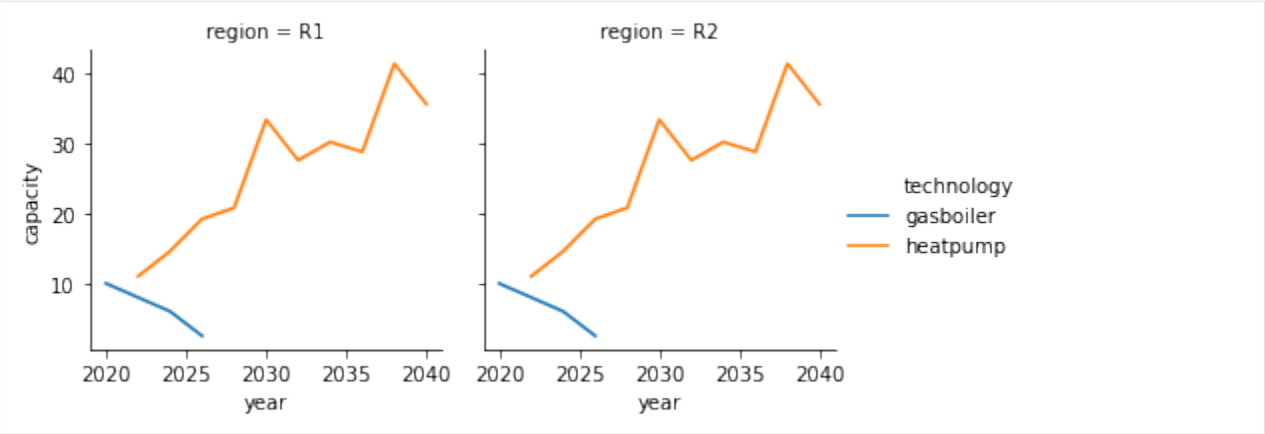
```
gas sector:
```




```
power sector:
```




```
residential sector:
```



Through the addition of more benchmark years, we are able to see a different scenario develop.

6.4.3 Next steps

In the next section we detail how to add an exogenous service demand, such as demand for heating or cooking.

6.5 Adding a service demand

In this section, we detail how to add a service demand to MUSE.

In the residential sector, a service demand could be cooking. Houses require energy to cook food and a technology to service this demand, such as an electric stove.

This process consists of setting a demand, either through inputs derived from the user or correlations of GDP and population which reflect the socio-economic development of a region or country. In addition, a technology must be added to service this new demand.

6.5.1 Addition of cooking demand

Firstly, we must add the demand section. In this example, we will add a cooking preset demand. To achieve this, we will now edit the `Residential2020Consumption.csv` and `Residential2050Consumption.csv` files, found within the `technodata/preset/` directory.

The `Residential2020Consumption.csv` file allows us to specify the demand in 2020 for each region and technology per timeslice. The `Residential2050Consumption.csv` file does the same but for the year 2050. The datapoints between these are interpolated.

Firstly, we must add the new service demand: `cook` as a column in these two files. Next, we add the demand. Again, the modified entries are in bold:

	RegionName	ProcessName	Timeslice	electricity	gas	heat	CO2f	wind	**cook**
0	R1	gasboiler	1	0	0	1	0	0	**1**
1	R1	gasboiler	2	0	0	1.5	0	0	**2**
2	R1	gasboiler	3	0	0	1	0	0	**1**
3	R1	gasboiler	4	0	0	1.5	0	0	**1.5**
4	R1	gasboiler	5	0	0	3	0	0	**2**
5	R1	gasboiler	6	0	0	2	0	0	**3**
6	R1	gasboiler	7	0	0	2	0	0	**2**
7	R1	gasboiler	8	0	0	2	0	0	**3**
8	R2	gasboiler	1	0	0	1	0	0	**1**
9	R2	gasboiler	2	0	0	1.5	0	0	**1**
10	R2	gasboiler	3	0	0	1	0	0	**1**
11	R2	gasboiler	4	0	0	1.5	0	0	**1.5**
12	R2	gasboiler	5	0	0	3	0	0	**2**
13	R2	gasboiler	6	0	0	2	0	0	**2**
14	R2	gasboiler	7	0	0	2	0	0	**2.5**
15	R2	gasboiler	8	0	0	2	0	0	**2**

As can be seen, we only need to add a `cook` column in the file, as well as the demand level for each timeslice and each region. This can be seen through the addition of a positive number in the `cook` column.

The process is very similar for the `Residential2050Consumption.csv` file, however, for this example, we often placed larger numbers to indicate higher demand in 2050. For the complete file see the link here INCLUDE LINK HERE.

Next, we must edit the files within the `input` folder. For this, we must add the `cook` service demand to each of these files.

First, we will amend the `BaseYearExport.csv` and `BaseYearImport.csv` files. For this, we say that there is no import or export of the `cook` service demand. A brief example is outlined below for `BaseYearExport.csv`:

RegionName	Attribute	Time	electricity	gas	heat	CO2f	wind	solar	**cook**
Unit	•	Year	PJ	PJ	PJ	kt	PJ	PJ	**PJ**
R1	Exports	2010	0	0	0	0	0	0	**0**
…	…	…	…	…	…	…	…	…	**…**
R2	Exports	2100	0	0	0	0	0	0	**0**

The same is true for the `BaseYearImport.csv` file:

RegionName	Attribute	Time	electricity	gas	heat	CO2f	wind	solar	**cook**
Unit	•	Year	PJ	PJ	PJ	kt	PJ	PJ	**PJ**
R1	Imports	2010	0	0	0	0	0	0	**0**
…	…	…	…	…	…	…	…	…	**…**
R2	Imports	2100	0	0	0	0	0	0	**0**

Next, we must edit the `GlobalCommodities.csv` file. This is where we define the new commodity `cook`. It tells

MUSE the commodity type, name, emissions factor of CO2 and heat rate, amongst other things.

The example used for this tutorial is below:

Commodity	CommodityType	CommodityName	CommodityEmissionFactor_C O2	HeatRate	Unit
Electricity	Energy	electricity	0	1	PJ
Gas	Energy	gas	56.1	1	PJ
Heat	Energy	heat	0	1	PJ
Wind	Energy	wind	0	1	PJ
CO2fuelcomsbustion	Environmental	CO2f	0	1	kt
Solar	Energy	solar	0	1	PJ
Cook	**Energy**	**cook**	**0**	**1**	**PJ**

Finally, the `Projections.csv` file must be changed. This is a large file which details the expected cost of the technology in the first benchmark year of the simulation. Due to its size, we will only show two rows of the new column `cook`.

RegionName	Attribute	Time	...	**cook**
Unit	•	Year	...	**MUS$2010/PJ**
R1	CommodityPrice	2010	...	**100**
...	...	...	...	**...**
R2	CommodityPrice	2100	...	**100**

We set every price of cook to be `100MUS$2010/PJ`

6.5.2 Addition of cooking technology

Next, we must add a technology to service this new demand. This is achieved through a similar process as the section in the *"1. adding a new technology"* section. However, we must be careful to specify the end-use of the technology as `cook`.

For this example, we will add two competing technologies to service the cooking demand: `electric_stove` and `gas_stove` to the `Technodata.csv` file in `/technodata/residential/Technodata.csv`.

Again for the interests of space, we have omitted the existing `gasboiler` and `heatpump` technologies. But we copy the `gasboiler` row for `R1` and paste it for the new `electric_stove` for both `R1` and `R2`. For `gas_stove` we copy and paste the data for `heatpump` from region `R1` for both `R1` and `R2`.

An important modification, however, is specifying the end-use for these new technologies to be `cook` and not `heat`.

Process-Name	Region-Name	Time	Level	cap_par	…	Fuel	EndUse	Agent2
Unit	•	Year	•	MUS$2010/PJ_a		•	•	Retrofit
gasboiler	R1	2020	fixed	3.8	…	gas	heat	1
…	…	…	…	…	…	…	…	…
electric_stove	**R1**	**2020**	**fixed**	**3.8**	**…**	**electricity**	**cook**	**1**
electric_stove	**R2**	**2020**	**fixed**	**3.8**	**…**	**electricity**	**cook**	**1**
gas_stove	**R1**	**2020**	**fixed**	**8.8667**	**…**	**gas**	**cook**	**1**
gas_stove	**R2**	**2020**	**fixed**	**8.8667**	**…**	**gas**	**cook**	**1**

As can be seen we have added two technologies, in the two regions with different `cap_par` costs. We specified their respective fules, and the enduse for both is `cook`. For the full file please see here INSERT LINK HERE.

We must also add the data for these new technologies to the following files:

- `CommIn.csv`
- `CommOut.csv`
- `ExistingCapacity.csv`

This is largely a similar process to the tutorial shown in *"adding a new technology"*. We must add the input to each of the technologies (gas and electricity for `gas_stove` and `electric_stove` respectively), outputs of `cook` for both and the existing capacity for each technology in each region.

Due to the additional demand for gas and electricity brought on by the new `cook` demand, it is necessary to relax the growth constraints for `gassupply1` in the `technodata/gas/technodata.csv` file. For this example, we set this file as follows:

Process-Name	Region-Name	Time	…	MaxCapacityAddition	MaxCapacityGrowth	TotalCapacityLimit	…	Agent2
Unit	•	Year	…	PJ	%	PJ	…	Retrofit
gassupply1	R1	2020	…	**100**	**5**	**500**	…	1
gassupply1	R2	2020	…	**100**	**5**	**120**	…	1

To prevent repetition of the *"adding a new technology"* section, we will leave the full files here INSERT LINK HERE.

Again, we run the simulation with our modified input files using the following command, in the relevant directory:

```
python -m pip muse settings.toml
```

Once this has run we are ready to visualise our results.

[1]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

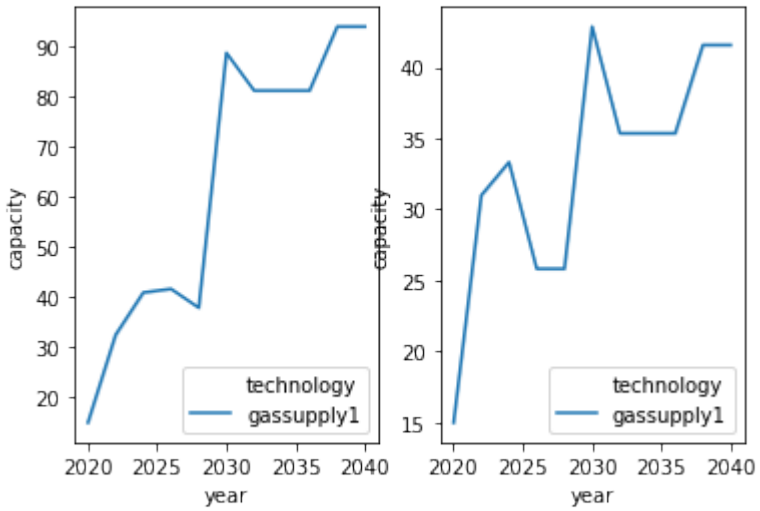
[2]:
```
mca_capacity = pd.read_csv("../tutorial-code/5-add-service-demand/Results/MCACapacity.csv
↪")
mca_capacity.head()
```

[2]:

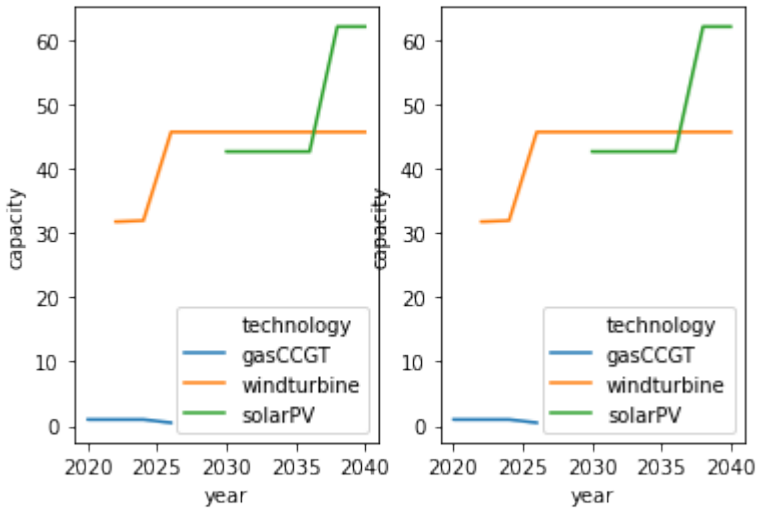
	technology	region	agent	type	sector	capacity	year
0	gas_stove	R1	A1	retrofit	residential	10.0	2020
1	gasboiler	R1	A1	retrofit	residential	10.0	2020
2	gas_stove	R2	A1	retrofit	residential	10.0	2020
3	gasboiler	R2	A1	retrofit	residential	10.0	2020
4	gas_stove	R1	A2	retrofit	residential	10.0	2020

[3]:
```
for name, sector in mca_capacity.groupby("sector"):
    print("{} sector:".format(name))
    fig, ax =plt.subplots(1,2)
    sns.lineplot(data=sector[sector.region=="R1"], x="year", y="capacity", hue=
↪"technology", ax=ax[0])
    sns.lineplot(data=sector[sector.region=="R2"], x="year", y="capacity", hue=
↪"technology", ax=ax[1])
    plt.show()
    plt.close()
```

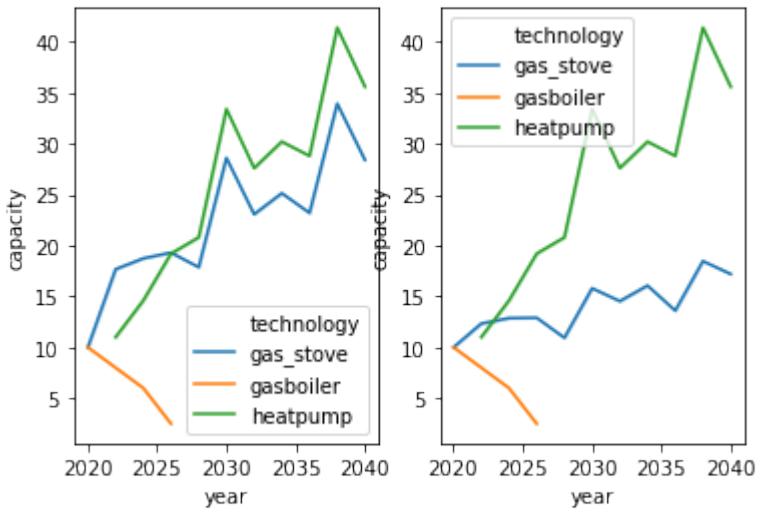
```
gas sector:
```




```
power sector:
```




```
residential sector:
```



We can see our new technology, the `gas_stove` is used and the `electric_stove` is not used at all. Therefore, there is an increase in `gassupply1` to accommodate for this growth in demand. However, this is not enough to displace `windturbine` by `gasCCGT` in the power sector.

6.5.3 Next steps

In the next section we will use a regression function to estimate electricity demand from GDP and population data.

6.6 Adding a service demand by correlation

In the previous section we added an exogenous service demand. That is, we explicitly specified what the demand would be per year.

However, we may not know what the electricity demand may be per year. Instead, we may conclude that our electricity demand is a function of the GDP and population of a particular region.

To accommodate such a scenario, MUSE enables us to choose a regression function that estimates service demands from GDP and population, which may be more certain in your case.

In this section we will show how this can be done.

6.6.1 Additional files

For this work, we will use the `default` example from the MUSE repository, and will not build on the previous examples. This is done to simplify the model at this point.

The full scenario files for the `default` example can be found here INSERT LINK HERE. We recommend that you download these files and save them to a location convenient to you, as we will be amending these throughout this tutorial.

Similarly to before, we must amend the `preset` folder for this. However, we no longer require the `Residential2020Consumption.csv` and `Residential2050Consumption.csv` files. These files set the exogenous service demand for the residential sector.

We must replace these files, with the following files:

- A macrodrivers file. This contains the drivers of the service demand that we want to model. For this example, these will include GDP based on purchasing power parity (GDP PPP) and the population that we expect from 2010 to 2110.
- A regression parameters file. This file will set the function type we would like to use to predict the service demand and the respective parameters of this regression file per region.
- A timeslice share file. This file sets how the demand is shared between timeslice.

The example files for each of those just mentioned can be found below, respectively:

- Macrodrivers.csv
- regressionparameters.csv
- TimesliceSharepreset.csv

For a full introduction to these files, see the link *here*.

Download these files and save them within the `preset` folder.

Next, we must amend our toml file to include our new way of calculating the preset service demand.

6.6.2 TOML file

Editting the TOML file to include this can be done relatively quickly if we know the variable names.

In the second bottom section of the toml file, you will see the following section:

```
[sectors.residential_presets]
type = 'presets'
priority = 0
consumption_path= "{path}/technodata/preset/*Consumption.csv"
```

This enables us to run the model in exogenous mode, but now we would like to run the model from the files previously mentioned. This can be done by linking new variables to the new files, as follows:

```
[sectors.residential_presets]
type = 'presets'
priority = 0

timeslice_shares_path = '{path}/technodata/preset/TimesliceSharepreset.csv'
macrodrivers_path = '{path}/technodata/preset/Macrodrivers.csv'
regression_path = '{path}/technodata/preset/regressionparameters.csv'
```

We effectively linked the new files to MUSE.

6.6.3 Increasing capacity constraints

Again, we must increase the capacity constraints. This is because the data in our GDP PPP and population files create a much higher demand than our previous toy example, due to the fact that it is more realistic for our particular region.

To ensure that we don't encounter any problems, we will relax our capacity constraints considerably.

For the full files see here. Make sure to take note of the columns:

- MaxCapacityAddition
- MaxCapacityGrowth
- TotalCapacityLimit

for the `Technodata.csv` files for each of the sectors.

6.6.4 Running and visualising our new results

With those changes made, we are now able to run our modified model, with the `python -m muse settings.toml` command in anaconda prompt, as before.

As before, we will now visualise the output.

[1]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

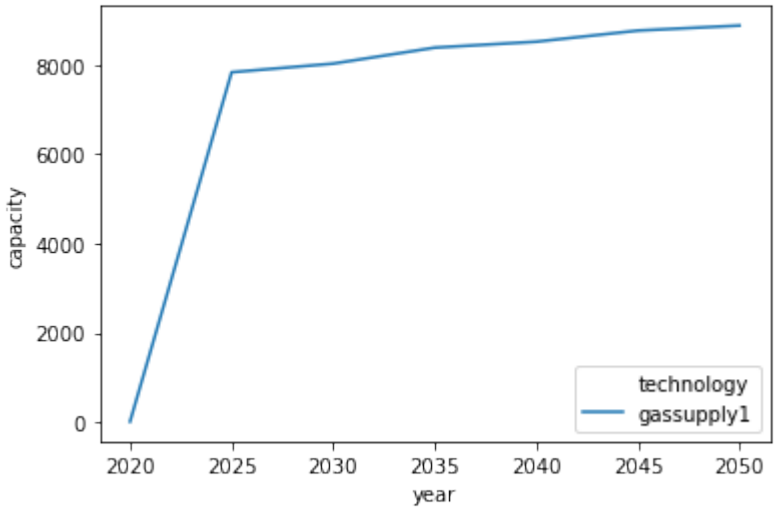
[3]:
```
mca_capacity = pd.read_csv("../tutorial-code/5-add-service-demand/2-correlation-demand/
↪Results/MCACapacity.csv")
mca_capacity.head()
```

[3]:

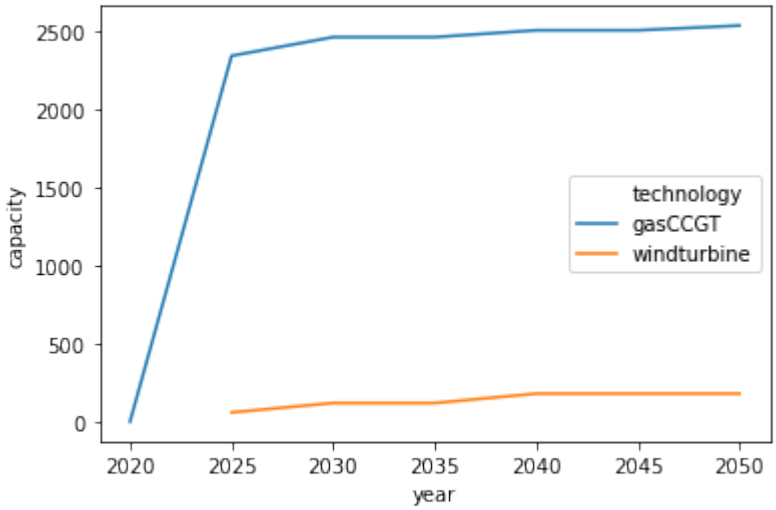
	technology	region	agent	type	sector	capacity	year
0	gasboiler	R1	A1	retrofit	residential	10.0000	2020
1	gasCCGT	R1	A1	retrofit	power	1.0000	2020
2	gassupply1	R1	A1	retrofit	gas	15.0000	2020
3	gasboiler	R1	A1	retrofit	residential	3044.3455	2025
4	heatpump	R1	A1	retrofit	residential	5323.8382	2025

[4]:
```
for name, sector in mca_capacity.groupby("sector"):
    print("{} sector:".format(name))
    sns.lineplot(data=sector[sector.region=="R1"], x="year", y="capacity", hue=
↪"technology")
    plt.show()
```

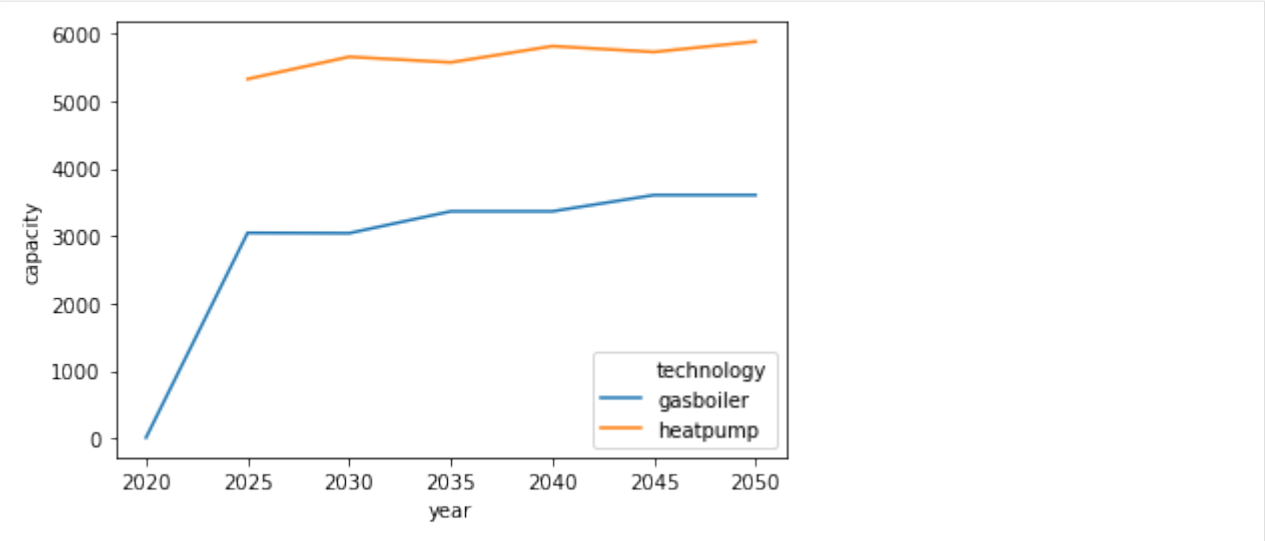
```
gas sector:
```




```
power sector:
```




```
residential sector:
```



As expected, we see a scenario emerge with much higher capacity limits. The demand does not increase linearly, with variations in the total demand in the residential sector. This is due to the new function not being a linear function.

6.6.5 Next steps

In the next section we will see how we can enforce outputs of technologies by timeslice. For instance, we can prevent solar photovoltaics from producing electricity at night, or ensure that a nuclear power plant runs at a minimum capacity during the day.

6.7 Production constraints by timeslice

In some sectors it may be the case that a technology can only output a certain amount at a certain time. For instance, solar photovoltaics (PV) don't produce power in the dark, and thus their output is limited at night.

In this section, we explain how to add constraints to outputs of technologies at certain timeslices. This could either by a maximum constraint, for instance with the solar PV example previously mentioned. Or, this could be a minimum constraint, where we expect a minimum amount of output by a nuclear power plant at all times.

6.7.1 Minimum timeslice

In this tutorial we will be amending the default example, which you can find here. Firstly, we will be imposing a minimum service factor for gasCCGT in the power sector. This is the minimum that a technology can output per timeslice.

To do this, we will need to create a new `csv` file that specifies the minimum service factor per timeslice.

An example of the file can be seen below, and downloaded here.

ProcessName	RegionName	Time	month	day	hour	UtilizationFactor	MinimumServiceFactor
Unit	•	Year	•	•	•	•	•
gasCCGT	R1	2020	all-year	all-week	night	1	1
gasCCGT	R1	2020	all-year	all-week	morning	1	2
gasCCGT	R1	2020	all-year	all-week	afternoon	1	3
gasCCGT	R1	2020	all-year	all-week	early-peak	1	2
gasCCGT	R1	2020	all-year	all-week	late-peak	1	5
gasCCGT	R1	2020	all-year	all-week	evening	1	6
windturbine	R1	2020	all-year	all-week	night	1	1
windturbine	R1	2020	all-year	all-week	morning	1	1
windturbine	R1	2020	all-year	all-week	afternoon	1	1
windturbine	R1	2020	all-year	all-week	early-peak	1	1
windturbine	R1	2020	all-year	all-week	late-peak	1	1
windturbine	R1	2020	all-year	all-week	evening	1	1

Notice that we have to specify the following columns: `ProcessName`, `RegionName`, `Time`, `month`, `day`, `hour`, `UtilizationFactor`, `MinimumServiceFactor`

The majority of these columns are self explanatory, and correspond to the columns in other `csv` files - for instance, `ProcessName`, `RegionName` and `Time`. The timeslice based columns, however, are dynamic and will match the levels as defined in the `toml` file.

The `UtilizationFactor` column specifies the maximum utilization factor for the respective technologies in the respective timeslices, and the `MinimumServiceFactor` specifies the minimum service factor of a technology.

Next, we must link this file to the `settings.toml` file. This is done by modifying the respective section. As we are modifying the power sector, we have to add it to the following section:

```
[sectors.power]
type = 'default'
priority = 2
dispatch_production = 'costed'

technodata = '{path}/technodata/power/Technodata.csv'
commodities_in = '{path}/technodata/power/CommIn.csv'
commodities_out = '{path}/technodata/power/CommOut.csv'
technodata_timeslices = '{path}/technodata/power/TechnodataTimeslices.csv'
```

Notice the technodata_timeslices path in the bottom row of the code above.

For the next part, we want to visualise the output of supply per technology, per timeslice. To do so, we will need to create our own output function. Creating our own output functions will be explored in later tutorials, so don't worry about understanding the code posted below for now.

Just copy and paste the code below into a python file called `output.py` and link to it in your `settings.toml` file as shown below:

```
plugins = "{path}/output.py"
```

[1]:
```
from typing import List, Optional, Text

import xarray as xr

from muse.outputs.sector import register_output_quantity
from muse.outputs.sector import market_quantity


@register_output_quantity
def supply_timeslice(
    market: xr.Dataset,
    capacity: xr.DataArray,
    technologies: xr.Dataset,
    sum_over: Optional[List[Text]] = None,
    drop: Optional[List[Text]] = None,
    rounding: int = 4,
) -> xr.DataArray:
    """Current supply."""
    market = market.reset_index("timeslice")
    result = (
        market_quantity(market.supply, sum_over=sum_over, drop=drop)
        .rename("supply")
        .to_dataframe()
        .round(rounding)
    )
    return result[result.supply != 0]
```

To link to our new output `supply_timeslice` we must past the following code in our `settings.toml` file.

```
[[sectors.power.outputs]]
filename = '{cwd}/{default_output_dir}/{Sector}/{Quantity}/{year}{suffix}'
quantity = "supply_timeslice"
sink = "csv"
overwrite = true
```

Once this has been completed, we are able to run MUSE as before, with the following command:

```
python -m muse settings.toml
```

Next, we will visualise the output of the technologies as before. However, this time we will visualise the supply outputs that we created with the previous function per technology and per timeslice.

[2]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

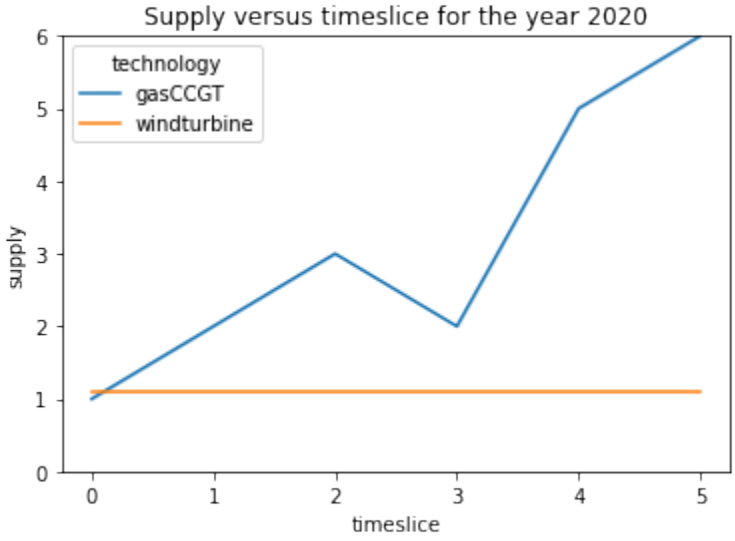
[5]:
```
power_supply_2020 = pd.read_csv("../tutorial-code/7-min-max-timeslice-constraints/1-min-
↪constraint/Results/Power/Supply_Timeslice/2020.csv")
sns.lineplot(data=power_supply_2020[power_supply_2020.commodity=="electricity"], x=
↪"timeslice", y="supply", hue="technology")
plt.title("Supply versus timeslice for the year 2020")
```

(continues on next page)

(continued from previous page)

```
plt.ylim(0,6)
```

```
[5]: (0.0, 6.0)
```



As expected, for the year 2020, the gasCCGT supplies the electricity mix by the minimum per timeslice. This is as specified in the `TechnodataTimeslices.csv` file. Starting from a supply of 1 in the first timeslice and ending with a supply of 6 in the last timeslice. The supply increases linearly apart from during the 3rd timeslice, where it reduces to 2.

6.7.2 Maximum timeslice constraint

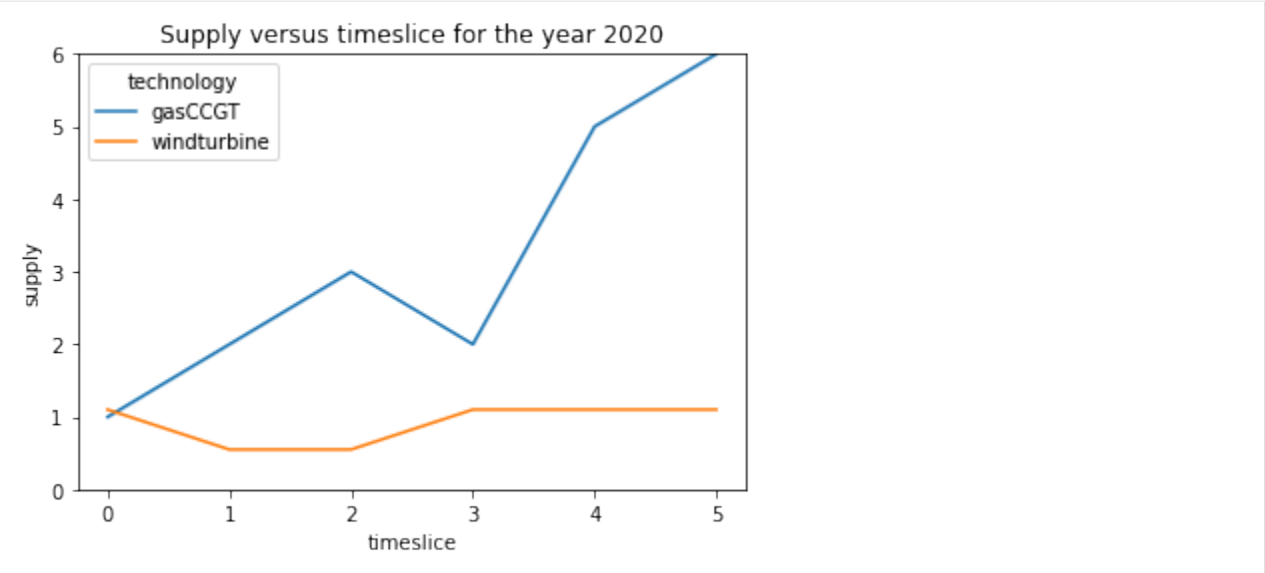
Next, we want to ensure that the supply of `windturbine` does not exceed a certain value during the day. This may be because, for example, there is reduced wind during the day. We will, therefore, modify the `TechnodataTimeslices.csv` file by changing the values of `UtilizationFactor`.

ProcessName	RegionName	Time	month	day	hour	UtilizationFactor	MinimumServiceFactor
Unit	•	Year	•	•	•	•	•
gasCCGT	R1	2020	all-year	all-week	night	1	1
gasCCGT	R1	2020	all-year	all-week	morning	1	2
gasCCGT	R1	2020	all-year	all-week	afternoon	1	3
gasCCGT	R1	2020	all-year	all-week	early-peak	1	2
gasCCGT	R1	2020	all-year	all-week	late-peak	1	5
gasCCGT	R1	2020	all-year	all-week	evening	1	6
windturbine	R1	2020	all-year	all-week	night	1	1
windturbine	R1	2020	all-year	all-week	morning	0.5	1
windturbine	R1	2020	all-year	all-week	afternoon	0.5	1
windturbine	R1	2020	all-year	all-week	early-peak	1	1
windturbine	R1	2020	all-year	all-week	late-peak	1	1
windturbine	R1	2020	all-year	all-week	evening	1	1

Once this has been saved, we can run the model again (`python -m muse settings.toml`). Next, we can visualise our results as before. We should hopefully see a reduction in the output of `windturbine` to 0.5 in the 2nd and 3rd timeslices:

[6]:
```
power_supply_2020 = pd.read_csv("../tutorial-code/7-min-max-timeslice-constraints/2-max-
↪constraint/Results/Power/Supply_Timeslice/2020.csv")
sns.lineplot(data=power_supply_2020[power_supply_2020.commodity=="electricity"], x=
↪"timeslice", y="supply", hue="technology")
plt.title("Supply versus timeslice for the year 2020")
plt.ylim(0,6)
```

[6]:
```
(0.0, 6.0)
```



As expected, we can see an enforced reduction in `windturbine` output to 0.5 in the 2nd (1) and 3rd (2) timeslices.

6.7.3 Next steps

This brings us to the end of the user guide! Using the information explained in this tutorial, or following similar steps, you will be able to create complex scenarios of your choosing.

For the full code to generate the final results, see here INSERT LINK HERE.

6.8 Tutorial Information

Below is a table to help you keep up with the different scenarios. The "Tutorial" header refers to the tutorial number as per the contents table. For example, Tutorial 1 is equal to "1. Adding a new technology". The "Agents", "Regions", "Residential Technologies", "Power Technologies" and "Gas Technologies" headers contain a tuple which contains the respective technologies present in the tutorial. The "Code" header provides a link to the files required to generate the tutorials.

In the table below, you will notice a x2 after some of the technology tuples. This refers to the fact that there are two regions, and the technologies within each region are the same for both regions.

Table 1: Scenarios

Tutorial	Agents	Regions	Residential Technologies	Power Technologies	Gas Technologies	Code
1	[A1]	[R1]	[gasboiler, heatpump]	[gasCCGT, windturbine, solarPV]	[gassuply1]	[1]
2	[A1, A2]	[R1]	[gasboiler, heatpump]	[gasCCGT, windturbine, solarPV]	[gassuply1]	[2]
3	[A1, A2] x2	[R1, R2]	[gasboiler, heatpump] x2	[gasCCGT, windturbine, solarPV] x2	[gassuply1] x2	[3]
4	[A1, A2] x2	[R1, R2]	[gasboiler, heatpump] x2	[gasCCGT, windturbine, solarPV] x2	[gassuply1] x2	[4]
5	[A1, A2] x2	[R1, R2]	[gasboiler, heatpump, electric_stove, gas_stove] x2	[gasCCGT, windturbine, solarPV] x2	[gassuply1] x2	[5]
6	[A1]	[R1]	[gasboiler, heatpump]	[gasCCGT, windturbine, solarPV]	[gassuply1]	[6]
7	[A1]	[R1]	[gasboiler, heatpump]	[gasCCGT, windturbine]	[gassuply1]	[7]

CHAPTER

SEVEN

INPUT FILES

In this section we detail each of the files required to run MUSE. We include information based on how these files should be used, as well as the data that populates them.

7.1 TOML primer

The full specification for TOML files can be found here. A TOML file is separated into sections, with each section except the topmost introduced by a name in square brackets. Sections can hold key-value pairs, e.g. a name associated with a value. For instance:

```
general_string_attribute = "x"

[some_section]
section_attribute = 12

[some_section.subsection]
subsetion_attribute = true
```

TOML is quite flexible in how one can define sections and attributes. The following three examples are equivalent:

```
[sectors.residential.production]
name = "match"
costing = "prices"
```

```
[sectors.residential]
production = {"name": "match", "costing": "prices"}
```

```
[sectors.residential]
production.name = "match"
production.costing = "prices"
```

Additionally, TOML files can contain tabular data, specified row-by-row using double square bracket. For instance, below we define a table with two rows and a single *column* called *some_table_of_data* (though column is not quite the right term, TOML tables are made more flexible than most tabular formats. Rather, each row can be considered a dictionary).

```
[[some_table_of_data]]
a_key = "a value"
```

(continues on next page)

(continued from previous page)

```
[[some_table_of_data]]
a_key = "another value"
```

As MUSE requires a number of data file, paths to files can be formated in a flexible manner. Paths can be formatted with shorthands for specific directories and are defined with curly-brackets. For example:

```
projection = '{path}/inputs/projection.csv'
timeslices_path = '{cwd}/technodata/timeslices.csv'
consumption_path = '{muse_sectors}/technodata/timeslices.csv'
```

path refers to the directory where the TOML file is located

cwd refers to the directory from which the muse simulation is launched

muse_sectors refers to the directory where default sectoral data is located

7.2 Simulation settings

This section details the TOML input for MUSE. The format for TOML files is described in this *previous section*. Here, however, we focus on sections and attributes that are specific to MUSE.

The TOML file can be read using `read_settings()`. The resulting data is used to construt the market clearing algorithm directly in the `MCA's factory function`.

7.2.1 Main section

This is the topmost section. It contains settings relevant to the simulation as a whole.

```
time_framework = [2020, 2025, 2030, 2035, 2040, 2045, 2050]
regions = ["USA"]
interpolation_mode = 'Active'
log_level = 'info'

equilibrium_variable = 'demand'
maximum_iterations = 100
tolerance = 0.1
tolerance_unmet_demand = -0.1
```

time_framework Required. List of years for which the simulation will run.

region Subset of regions to consider. If not given, defaults to all regions found in the simulation data.

interpolation_mode interpolation when reading the initial market. One of *linear*, *nearest*, *zero*, *slinear*, *quadratic*, *cubic*. Defaults to *linear*.

- *linear* interpolates using a linear function.
- *nearest* uses nearest-neighbour interpolation. That is, the closest known data point is used as the prediction for the data point to interpolate.
- *zero* assumes that the data point to interpolate is zero.
- *slinear* uses a spline of order 1. This is a similar method to *linear* interpolation.
- *quadratic* uses a quadratic equation for interpolation. This should be used if you expect your data to take a quadratic form.

- *cubic* interpolation is similar to *quadratic* interpolation, but uses a cubic function for interpolation.

log_level: verbosity of the output.

equilibirum_variable whether equilibrium of *demand* or *prices* should be sought. Defaults to *demand*.

maximum_iterations Maximum number of iterations when searching for equilibrium. Defaults to 3.

tolerance Tolerance criteria when checking for equilibrium. Defaults to 0.1. 0.1 signifies that 10% of a deviation is allowed among the iterative value of either demand or price over a year per region.

tolerance_unmet_demand Criteria checking whether the demand has been met. Defaults to -0.1.

excluded_commodities List of commodities excluded from the equilibrium considerations. Defaults to the list *["CO2f", "CO2r", "CO2c", "CO2s", "CH4", "N2O", "f-gases"]*.

plugins Path or list of paths to extra python plugins, i.e. files with registered functions such as `register_output_quantity()`.

7.2.2 Carbon market

This section containts the settings related to the modelling of the carbon market. If omitted, it defaults to not including the carbon market in the simulation.

Example

```
[carbon_budget_control]
budget = []
```

budget Yearly budget. There should be one item for each year the simulation will run. In other words, if given and not empty, this is a list with the same length as *time_framework* from the main section. If not given or an empty list, then the carbon market feature is disabled. Defaults to an empty list.

method Method used to equilibrate the carbon market. The only preset option is *fitting*, however this can be expanded with the *@register_carbon_budget_fitter* hook in *muse.carbon_budget*.

The market-clearing algortihm iterates over the sectors until the market reaches an equilibrium in the foresight period (the period next to the one analysed). This is represented by a stable variation of a commodity demand (or price) between iterations below a defined tolerance. The market-clearing algorithm samples a user-defined set of carbon prices and estimates a regression model of the global emissions as a function of the carbon price. The regression model is set by the user. The new carbon price is estimated as a root of the regression model estimated at the value of the emission equal to the user-defined emission cap in the foresight period.

commodities Commodities that make up the carbon market. Defaults to an empty list.

control_undershoot Whether to control carbon budget undershoots. This parameter allows for carbon tax credit from one year to be passed to the next in the case of less carbon being emitted than the budget. Defaults to True.

control_overshoot Whether to control carbon budget overshoots. If the amount of carbon emitted is above the carbon budget, this parameter specifies whether this deficit is carried over to the next year. Defaults to True.

method_options: Additional options for the specific carbon method.

fitter specifies the regression model fit. Predefined options are *linear* and *exponential*. Further options can be defined using the *@register_carbon_budget_fitter* hook in *muse.carbon_budget*.

7.2.3 Global input files

Defines the paths specific simulation data files. The paths can be formatted as explained in the *TOML primer*.

```
[global_input_files]
projections = '{path}/inputs/Projections.csv'
regions = '{path}/inputs/Regions.csv'
global_commodities = '{path}/inputs/MUSEGlobalCommodities.csv'
```

projections: Path to a csv file giving initial market projection. See *Initial Market Projection*.

regions: Path to a csv file describing the regions. See *Regional data*.

global_commodities: Path to a csv file describing the comodities in the simulation. See *Commodity Description*.

7.2.4 Timeslices

Time-slices represent a sub-year disaggregation of commodity demand. Generally, timeslices are expected to introduce several levels, e.g. season, day, or hour. The simplest is to show the TOML for the default timeslice:

```
[timeslices]
winter.weekday.night = 396
winter.weekday.morning = 396
winter.weekday.afternoon = 264
winter.weekday.early-peak = 66
winter.weekday.late-peak = 66
winter.weekday.evening = 396
winter.weekend.night = 156
winter.weekend.morning = 156
winter.weekend.afternoon = 156
winter.weekend.evening = 156
spring-autumn.weekday.night = 792
spring-autumn.weekday.morning = 792
spring-autumn.weekday.afternoon = 528
spring-autumn.weekday.early-peak = 132
spring-autumn.weekday.late-peak = 132
spring-autumn.weekday.evening = 792
spring-autumn.weekend.night = 300
spring-autumn.weekend.morning = 300
spring-autumn.weekend.afternoon = 300
spring-autumn.weekend.evening = 300
summer.weekday.night = 396
summer.weekday.morning  = 396
summer.weekday.afternoon = 264
summer.weekday.early-peak = 66
summer.weekday.late-peak = 66
summer.weekday.evening = 396
summer.weekend.night = 150
summer.weekend.morning = 150
summer.weekend.afternoon = 150
summer.weekend.evening = 150
level_names = ["month", "day", "hour"]
```

This input introduces three levels, via `level_names`: `month`, `day`, `hours`. Other simulations may want fewer or more levels. The `month` level is split into three points of data, `winter`, `spring-autumn`, `summer`. Then `day` splits out

weekdays from weekends, and so on. Each line indicates the number of hours for the relevant slice. It should be noted that the slices are not a cartesian products of each levels. For instance, there no `peak` periods during weekends. All that matters is that the relative weights (i.e. the number of hours) are consistent and sum up to a year.

The input above defines the finest times slice in the code. In order to define rougher timeslices we can introduce items in each levels that represent aggregates at that level. By default, we have the following:

```
[timeslices.aggregates]
all-day = ["night", "morning", "afternoon", "early-peak", "late-peak", "evening"]
all-week = ["weekday", "weekend"]
all-year = ["winter", "summer", "spring-autumn"]
```

Here, `all-day` aggregates the full day. However, one could potentially create aggregates such as:

```
[timeslices.aggregates]
daylight = ["morning", "afternoon", "early-peak", "late-peak"]
nightlife = ["evening", "night"]
```

Once the finest timeslice and its aggregates are given, it is possible for each sector to define the timeslice simply by refering to the slices it will use at each level.

```
[sectors.some_sector.timeslice_levels]
day = ["daylight", "nightlife"]
month = ["all-year"]
```

Above, `sectors.some_sector.timeslice_levels.week` defaults its value in the finest timeslice. Indeed, if the subsection `sectors.some_sector.timeslice_levels` is not given, then the sector will default to using the finest timeslices.

Similarly, it is possible to specify a timeslice for the mca by adding an *mca.timeslice_levels* section. However, be aware that if the MCA uses a rougher timeslice framework, the market will be expressed within it. Hence information from sectors with a finer timeslice framework will be lost.

7.2.5 Standard sectors

Sectors are declared in the TOML file by adding a subsection to the *sectors* section:

```
[sectors.residential]
type = 'default'
[sectors.power]
type = 'default'
```

Above, we've added two sectors, residential and power. The name of the subsection is only used for identification. In other words, it should be chosen to be meaningful to the user, since it will not affect the model itself.

Sectors are defined in `Sector`.

A sector accepts a number of attributes and subsections.

type Defines the kind of sector this is. *Standard* sectors are those with type "default". This value corresponds to the name with which a sector class is registerd with MUSE, via `register_sector()`. [INSERT OTHER OPTIONS HERE]

priority An integer denoting which sectors runs when. Lower values imply the sector will run earlier. If two sectors share the same priority. Later sectors can depend on earlier sectors for the their input. If two sectors share the same priority, then their order is not defined. Indeed, it should indicate that they can run in parallel. For simplicity, the keyword also accepts standard values:

- "preset": 0
- "demand": 10
- "conversion": 20
- "supply": 30
- "last": 100

Defaults to "last".

interpolation Interpolation method user when filling in missing values. Available interpolation methods depend on the underlying scipy method's kind attribute.

investment_production In its simplest form, this is the name of a method to compute the production from a sector, as used when splitting the demand across agents. In other words, this is the computation of the production which affects future investments. In it's more general form, *production* can be a subsection of its own, with a "name" attribute. For instance:

```
[sectors.residential.production]
name = "match"
costing = "prices"
```

MUSE provides two methods in `muse.production`:

- **share: the production is the maximum production for the existing capacity and** the technology's utilization factor. See `muse.production.maximum_production()`.
- **match: production and demand are matched according to a given cost metric. The** cost metric defaults to "prices". It can be modified by using the general form given above, with a "costing" attribute. The latter can be "prices", "gross_margin", or "lcoe". See `muse.production.demand_matched_production()`.

production can also refer to any custom production method registered with MUSE via `muse.production.register_production()`.

Defaults to "share".

dispatch_production The name of the production method used to compute the sector's output, as returned to the muse market clearing algorithm. In other words, this is computation of the production method which will affect other sectors.

It has the same format and options as the *production* attribute above.

demand_share A method used to split the MCA demand into seperate parts to be serviced by specific agents. There is currently only one option, "new_and_retro", corresponding to *new* and *retro* agents.

interactions Defines interactions between agents. These interactions take place right before new investments are computed. The interactions can be anything. They are expected to modify the agents and their assets. MUSE provides a default set of interactions that have *new* agents pass on their assets to the corresponding *retro* agent, and the *retro* agents pass on the make-up of their assets to the corresponding *new* agents.

interactions are specified as a *TOML array*, e.g. with double brackets. Each sector can specify an arbitrary number of interactaction, simply by adding an extra interaction row.

There are two orthogonal concepts to interactions:

- a *net* defines the set of agents that interact. A set can contain any number of agents, whether zero, two, or all agents in a sector. See `muse.interactions.register_interaction_net()`.
- an *interaction* defines how the net actually interacts. See `muse.interactions.register_agent_interaction()`.

In practice, we always consider sequences of nets (i.e. more than one net) that interact using the same interaction function.

Hence, the input looks something like the following:

```
[[sectors.commercial.interactions]]
net = 'new_to_retro'
interaction = 'transfer'
```

"new_to_retro" is a function that figures out all "new/retro" pairs of agents. Whereas "transfer" is a function that performs the transfer of assets and information between each pair.

Furthermore, it is possible to pass parameters to either the net of the interaction as follows:

```
[[sectors.commercial.interactions]]
net = {"name": "some_net", "param": "some value"}
interaction = {"name": "some_interaction", "param": "some other value"}
```

The parameters will depend on the net and interaction functions. Neither "new_to_retro" nor "transfer" take any arguments at this point. MUSE interaction facilities are defined in `muse.interactions`.

output Outputs are made up of several components. MUSE is designed to allow users to mix-and-match both how and what to save.

output is specified as a TOML array, e.g. with double brackets. Each sector can specify an arbitrary number of outputs, simply by adding an extra output row.

A single row looks like this:

```
[[sectors.commercial.outputs]]
filename = '{cwd}/Results/{Sector}/{Quantity}/{year}{suffix}'
quantity = "capacity"
sink = 'csv'
overwrite = true
```

The following attributes are available:

- **quantity: Name of the quantity to save. Currently, only *capacity* exists,** refering to `muse.outputs.capacity()`. However, users can customize and create further output quantities by registering with MUSE via `muse.outputs.register_output_quantity()`. See `muse.outputs` for more details.
- **sink: the sink is the place (disk, cloud, database, etc...) and format with which** the computed quantity is saved. Currently only sinks that save to files are implemented. The filename can specified via *filename*, as given below. The following sinks are available: "csv", "netcfd", "excel". However, more sinks can be added by interested users, and registered with MUSE via `muse.outputs.register_output_sink()`. See `muse.outputs` for more details.
- **filename: defines the format of the file where to save the data. There are several** standard values that are automatically substituted:
  - cwd: current working directory, where MUSE was started
  - path: directory where the TOML file resides
  - sector: name of the current sector (.e.g. "commercial" above)
  - Sector: capitalized name of the current sector
  - quantity: name of the quantity to save (as given by the quantity attribute)
  - Quantity: capitablized name of the quantity to save

  - year: current year
  - suffix: standard suffix/file extension of the sink

  Defaults to *{cwd}/{default_output_dir}/{Sector}/{Quantity}/{year}{suffix}*.

- **overwrite: If *False* MUSE will issue an error and abort, instead of** overwriting an existing file. Defaults to *False*. This prevents important output files from being overwritten.

technodata Path to a csv file containing the characterization of the technologies involved in the sector, e.g. lifetime, capital costs, etc... See *Techno-data Timeslices*.

timeslice_levels Slices to consider in a level. If absent, defaults to the finest timeslices. See *Timeslices*

commodities_in Path to a csv file describing the inputs of each technology involved in the sector. See *General features*.

commodities_out Path to a csv file describing the outputs of each technology involved in the sector. See *Output Commodities*.

existing_capacity Path to a csv file describing the initial capacity of the sector. See *Existing Sectoral Capacity*.

agents Path to a csv file describing the agents in the sector. See *Agents*.

7.2.6 Preset sectors

The commodity production, commodity consumption and product prices of preset sectors are determined exogeneously. They are know from the start of the simulation and are not affected by the simulation.

Preset sectors are defined in `PresetSector`.

The three components, production, consumption, and prices, can be set independantly and not all three need to be set. Production and consumption default to zero, and prices default to leaving things unchanged.

The following defines a standard preset sector where consumption is defined as a function of macro-economic data, i.e. population and gdp.

```
[sectors.commercial_presets]
type = 'presets'
priority = 'presets'
timeslice_shares_path = '{path}/technodata/TimesliceShareCommercial.csv'
macrodrivers_path = '{path}/technodata/Macrodrivers.csv'
regression_path = '{path}/technodata/regressionparameters.csv'
timeslices_levels = {'day': ['all-day']}
forecast = [0, 5]
```

The following attributes are accepted:

type: See the attribute in the standard mode, *type*. *Preset* sectors are those with type "presets".

priority See the attribute in the standard mode, *priority*.

timeslices_levels: See the attribute in the standard mode, *Timeslices*.

consumption_path: CSV output files, one per year. This attribute can include wild cards, i.e. '*', which can match anything. For instance: *consumption_path = "{cwd}/Consumtion*.csv"* will match any csv file starting with "Consumption" in the current working directory. The file names must include the year for which it defines the consumption, e.g. *Consumption2015.csv*.

The CSV format should follow the following format:

Table 1: Consumption

	RegionName	ProcessName	TimeSlice	electricity	diesel	algae
0	USA	fluorescent light	1	1.9	0	0
1	USA	fluorescent light	2	1.8	0	0

The index column as well as "RegionName", "ProcessName", and "TimeSlice" must be present. Further columns are reserved for commodities. "TimeSlice" refers to the index of the timeslice.

supply_path: CSV file, one per year, indicating the amount of a commodities produced. It follows the same format as *consumption_path*.

supply_path: CSV file, one per year, indicating the amount of a commodities produced. It follows the same format as *consumption_path*.

prices_path: CSV file indicating the amount of a commodities produced. The format of the CSV files follows that of *Initial Market Projection*.

demand_path: Incompatible with *consumption_path* or *macrodrivers_path*. A CSV file containing the consumption in the same format as *Initial Market Projection*.

macrodrivers_path: Incompatible with *consumption_path* or *demand_path*. Path to a CSV file giving the profile of the macrodrivers. Also requires *regression_path*.

regression_path: Incompatible with *consumption_path* or *demand_path*. Path to a CSV file giving the regression parameters with respect to the macrodrivers. Also requires *macrodrivers_path*.

timeslice_shares_path Optional csv file giving shares per timeslice. Requires *macrodrivers_path*.

filters: Optional dictionary of entries by which to filter the consumption. Requires *macrodrivers_path*. For instance,

```
filters.region = ["USA", "ASEA"]
filters.commodity = ["algae", "fluorescent light"]
```

7.2.7 Legacy Sectors

Legacy sectors wrap sectors developed for a previous version of MUSE to the open-source version.

Preset sectors are defined in `PresetSector`.

The can be defined in the TOML file as follows:

```
[global_input_files]
macrodrivers = '{path}/input/Macrodrivers.csv'
regions = '{path}/input/Regions.csv'
global_commodities = '{path}/input/MUSEGlobalCommodities.csv'

[sectors.Industry]
type = 'legacy'
priority = 'demand'
agregation_level = 'month'
excess = 0

userdata_path = '{muse_sectors}/Industry'
technodata_path = '{muse_sectors}/Industry'
timeslices_path = '{muse_sectors}/Industry/TimeslicesIndustry.csv'
output_path = '{path}/output'
```

For historical reasons, the three *global_input_files* above are required. The sector itself can use the following attributes.

type: See the attribute in the standard mode, *type*. *Legacy* sectors are those with type "legacy".

priority See the attribute in the standard mode, *priority*.

agregation_level: Information relevant to the sector's timeslice.

excess: Excess factor used to model early obsolescence.

timeslices_path: Path to a timeslice *time_slices*.

userdata_path: Path to a directory with sector-specific data files.

technodata_path: Path to a technodata CSV file. See. *Techno-data Timeslices*.

output_path: Path to a diretory where the sector will write output files.

7.3 Input Files

7.3.1 Initial Market Projection

MUSE needs an initial projection of the market prices for each period of the simulation.

- The price trajectory is needed if the MCA works in *equilibrium* mode as an initial start point for the base year of the simulation. The market will override the calculated prices obtained from each commodity equilibrium for all the future periods following the base year
- Similarly, if the market works in a *carbon budget* mode, the prices are used as a starting point. The only difference from the previous case is that the MCA calculates an additional global market price for carbon dioxide (and additional pollutants if required)
- If the MCA works in an *exogenous* mode, it will use the initial market projection as the projection for the the base year and all the future periods of the simulation

The forward price trajectory should follow the structure reported in the table below.

Table 2: Initial market projections

RegionName	Attribute	Time	com1	com2	com3
Unit	•	Year	MUS$2010/PJ	MUS$2010/PJ	MUS$2010/PJ
region1	CommodityPrice	2010	20	1.9583	2
region1	CommodityPrice	2015	20	1.9583	2
region1	CommodityPrice	2020	20.38518042	1.996014941	2.038518042
region1	CommodityPrice	2025	20.77777903	2.034456234	2.077777903
region1	CommodityPrice	2030	21.17793872	2.073637869	2.117793872
region1	CommodityPrice	2035	21.58580508	2.113574105	2.158580508
region1	CommodityPrice	2040	22.00152655	2.154279472	2.200152655
region1	CommodityPrice	2045	22.42525441	2.195768786	2.242525441
region1	CommodityPrice	2050	22.85714286	2.238057143	2.285714286

RegionName represents the region ID and needs to be consistent across all the data inputs

Attribute defines the attribute type. In this case it refers to the CommodityPrice; it is relevant only for internal use

Time corresponds to the time periods of the simulation; the simulated time framework in the example goes from 2010 through to 2050 with a 5-year time step

com1, . . . , comN Any further columns represent the commodities modelled, as defined in the global commodities the row Unit reports the unit in which the technology consumption is defined; it is for the user internal reference only. The names *comX* should be replaced with the names of the commodities.

7.3.2 Regional data

MUSE requires the definition of the methodology used for investment and dispatch and alias demand matching. The methodology has to be defined by region and subregion, meant as a geographical subdivision in a region. Currently, the methodology definition is important for the legacy sectors only, and can therefore be ignored for the open source version of MUSE.

Below the generic structure of the input commodity file for the electric heater is shown:

Table 3: Methodology used in investment and demand matching

SectorName	RegionName	Subregion	sMethodologyPlanning	sMethodologyDispatch
Agriculture	region1	region1	NPV	DCF
Bioenergy	region1	region1	NPV	DCF
Industry	region1	region1	NPV	DCF
Residential	region1	region1	EAC	EAC
Commercial	region1	region1	EAC	EAC
Transport	region1	region1	LCOE	LCOE
Power	region1	region1	LCOE	LCOE
Refinery	region1	region1	LCOE	LCOE
Supply	region1	region1	LCOE	LCOE

SectorName represents the sector_ID and needs to be consistent across the data input files

RegionName represents the region ID and needs to be consistent across all the data inputs

Subregion represents the subregion ID and needs to be consistent across all the data inputs

sMethodologyPlanning reports the cost quantity used for making investments in new technologies in each sector (e.g. NPV stands for net present value, EAC stands for equivalent annual costs, LCOE stands for levelised cost of energy)

sMethodologyDispatch reports the cost quantity used for the demand matching using existing technologies in each sector (e.g. DCF stands for discounted cash flow, EAC stands for equivalent annual cost, LCOE stands for levelised cost of energy)

7.3.3 Commodity Description

MUSE handles a configurable number and type of commodities which are primarily used to represent energy, services, pollutants/emissions. The commodities for the simulation as a whole are defined in a csv file with the following structure.

Table 4: Global commodities

Commodity	CommodityType	CommodityName	CommodityEmissionFactor_CO2	HeatRate	Unit
Coal	Energy	hardcoal	94.6	29	PJ
Agricultural-residues	Energy	agrires	112	15.4	PJ

Commodity represents the extended name of a commodity

CommodityType defines the type of a commodity (i.e. energy, material or environmental)

CommodityName is the internal name used for a commodity inside the model.

CommodityEmissionFactor_CO2 is CO2 emission per unit of commodity flow

HeatRate represents the lower heating value of an energy commodity

Unit is the unit used as a basis for all the input data. More specifically the model allows a totally flexible way of defining the commodities. CommodityName is currently the only column used internally as it defines the names of commodities and needs to be used consistently across all the input data files. The remaining columns of the file are only relevant for the user internal reference for the original sets of assumptions used.

7.3.4 Techno-data

The techno-data includes the techno-economic characteristics of each technology such as capital, fixed and variable cost, lifetime, utilisation factor. The techno-data should follow the structure reported in the table. The column order is not important and additional input data can alsobe read in this format. In the table, the electric boiler used in households is taken as an example for a generic region, region1.

Table 5: Techno-data

ProcessName	RegionName	Time	Level	cap_par	cap_exp	fix_par	...
resBoilerElectric	region1	2010	fixed	3.81	1.00	0.38	...
resBoilerElectric	region1	2030	fixed	3.81	1.00	0.38	...

ProcessName represents the technology ID and needs to be consistent across all the data inputs

RegionName represents the region ID and needs to be consistent across all the data inputs

Time represents the period of the simulation to which the value applies; it needs to contain at least the base year of the simulation

Level characterises either a fixed or a flexible input type

cap_par, cap_exp are used in the capital cost estimation. Capital costs are calculated as:

$$\text{CAPEX} = \text{cap\_par} * (\text{Capacity})^{\text{cap\_exp}}$$

where the parameter cap_par is estimated at a selected reference size (i.e. Capref), such as:

$$\text{cap\_par} = \left(\frac{\text{CAPEXref}}{\text{Capref}}\right)^{\text{cap\_exp}}$$

Capref is decided by the modeller before filling the input data files.

This allows the model to take into account economies of scale. ie. As *Capacity* increases, the price of the technology decreases. This does not include technological learning parameters, where prices may come down due to learning.

fix_par, fix_exp are used in the fixed cost estimation. Fixed costs are calculated as:

$$\text{FOM} = \text{fix\_par} * (\text{Capacity})^{\text{fix\_exp}}$$

where the parameter fix_par is estimated at a selected reference size (i.e. Capref), such as:

$$\text{fix\_par} = \left(\frac{\text{FOMref}}{\text{Capref}}\right)^{\text{fix\_exp}}$$

Capref is decided by the modeller before filling the input data files.

var_par, var_exp are used in the variable costs estimation. These variable costs are capacity dependent Variable costs are calculated as:

$$\text{VAREX} = \text{cap\_par} * (\text{Capacity})^{\text{cap\_exp}}$$

where the parameter var_par is estimated at a selected reference size (i.e. Capref), such as:

$$\text{var\_par} = \left(\frac{\text{VARref}}{\text{Capref}}\right)^{\text{var\_exp}}$$

Capref is decided by the modeller before filling the input data files.

MaxCapacityAddition represents the maximum addition of installed capacity per technology, per year in a period, per region.

MaxCapacityGrowth represents the percentage growth per year based on the available stock in a year, per region and technology.

TotalCapacityLimit represents the total capacity limit per technology, region and year.

TechnicalLife represents the number of years that a technology operates before it is decommissioned.

UtilizationFactor represents the maximum actual output of the technology in a year, divided by the theoretical maximum output if the technology were operating at full capacity for the whole year.

ScalingSize represents the minimum size of a technology to be installed.

efficiency is calculated as the ratio between the total output commodities and the input commodities.

Type defines the type of a technology. This variable is used for the search space in the agents csv file. It allows for the agents to filter for technologies of a similar type, for example.

Fuel defines the fuel used by a technology.

EndUse defines the end use of a technology.

InterestRate is the technology interest rate. This is used for the interest used in the discount rate.

Agent_0, . . . , Agent_N represent the allocation of the initial capacity to the each agent.

The input data has to be provided for the base year. Additional years within the time framework of the overall simulation can be defined. In this case, MUSE would interpolate the values between the provided periods and assume a constant value afterwards.

7.3.5 Techno-data Timeslices

The techno-data timeslices is an optinal file which contains information on technologies, their region, timeslices, utilization factors and minimum service factor. The objective of this file is to link the utilization factor and minimum service factor to timeslices. For instance, if you were to model solar photovoltaics, you would probably want to specify that they can not produce any electricty at night, or if you're modelling a nuclear power plant, that they must generate a minimum amount of electricity. The techno-data timeslice file enables you to do that. Note, that if this file is not present, the utilization facto will be used from the technodata file.

Table 6: Techno-data

Process-Name	Region-Name	Time	month	day	hour	Utiliza-tionFactor	Minimum-Service-Factor
Unit	•	Year	•	•	•	•	•
gasCCGT	R1	2020	all-year	all-week	night	1	1
gasCCGT	R1	2020	all-year	all-week	morning	1	2

ProcessName represents the technology ID and needs to be consistent across all the data inputs

RegionName represents the region ID and needs to be consistent across all the data inputs

Time represents the period of the simulation to which the value applies; it needs to contain at least the base year of the simulation

month represents the first level of the timeslice. This input is dynamic and so does not necessarily need to be a month, but a season for instance. As long as it matches with the toml file.

day represents the second level of the timeslice. Again, this input is dynamic, and thus does not necessarily need to be a day - as long as it matches with the toml file.

hour represents the third level of the timeslice.

UtilizationFactor represents the maximum actual output of the technology in a year, divided by the theoretical maximum output if the technology were operating at full capacity for the whole year per timeslice. This overwrites the UtilizationFactor in the technodata file.

MinimumServiceFactor represents the minimum service that a technology can output. For instance, the minimum amount of electricity that can be output from a nuclear power plant at a particular timeslice.

The input data has to be provided for the base year. Additional years within the time framework of the overall simulation can be defined. In this case, MUSE would interpolate the values between the provided periods and assume a constant value afterwards.

7.3.6 Time-slices

Note: This input file is only for legacy sectors, which are not part of the open source release. For time-slice settings in the open source release, please see the *Simulation settings*.

Time-slices represent a sub-year disaggregation of commodity demand. They are fully flexible in number and names as to serve the specific representation of the commodity demand, supply, and supply cost profile in each energy sector. Each time slice is independent in terms of the number of represent hours, as long as it is meaningful for the users and their data inputs. 1 is the minimum number of time-slice as this would correspond to a full year. The time-slice definition of a sector affects the commodity price profile and the supply cost profile.

The csv file for the time-slice definition would report the length (in hours) of each time slice as characteristic to the selected sector to represent diurnal, weekly and seasonal variation of energy commodities, demand and supply, as shown in the table for 30 time-slices.

Table 7: Time-slices

AgLevel	SN	Month	Day	Hour	RepresentHours
Hour	1	Winter	Weekday	Night	396
Hour	2	Winter	Weekday	Morning	396
Hour	3	Winter	Weekday	Afternoon	264
Hour	4	Winter	Weekday	EarlyPeak	66
Hour	5	Winter	Weekday	LatePeak	66
Hour	6	Winter	Weekday	Evening	396
Hour	7	Winter	Weekend	Night	156
Hour	8	Winter	Weekend	Morning	156
Hour	9	Winter	Weekend	Afternoon	156
Hour	10	Winter	Weekend	Evening	156
Hour	11	SpringAutumn	Weekday	Night	792
Hour	12	SpringAutumn	Weekday	Morning	792
Hour	13	SpringAutumn	Weekday	Afternoon	528
Hour	14	SpringAutumn	Weekday	EarlyPeak	132
Hour	15	SpringAutumn	Weekday	LatePeak	132
Hour	16	SpringAutumn	Weekday	Evening	792
Hour	17	SpringAutumn	Weekend	Night	300
Hour	18	SpringAutumn	Weekend	Morning	300
Hour	19	SpringAutumn	Weekend	Afternoon	300

continues on next page

Table 7 – continued from previous page

AgLevel	SN	Month	Day	Hour	RepresentHours
Hour	20	SpringAutumn	Weekend	Evening	300
Hour	21	Summer	Weekday	Night	396
Hour	22	Summer	Weekday	Morning	396
Hour	23	Summer	Weekday	Afternoon	264
Hour	24	Summer	Weekday	EarlyPeak	66
Hour	25	Summer	Weekday	LatePeak	66
Hour	26	Summer	Weekday	Evening	396
Hour	27	Summer	Weekend	Night	150
Hour	28	Summer	Weekend	Morning	150
Hour	29	Summer	Weekend	Afternoon	150
Hour	30	Summer	Weekend	Evening	150

It reports the aggregation level of the sector time-slices (AgLevel), slice number (SN), seasonal time slices (Month), weekly time slices (Day), hourly profile (Hour), the amount of hours associated to each time slice (RepresentHours).

7.3.7 Input Commodities

Input commodities are the commodities consumed (also called consumables in MUSE) by each technology. They are defined in a csv file which describes the commodity inputs to each technology, calculated per unit of technology activity. Where the unit is defined by the user (e.g. petajoules). See *below* for a description.

7.3.8 Output Commodities

Output commodities are the commodities produced (also called products in MUSE) by each technology. They are defined in a csv file which describes the commodity outputs from each technology, defined per unit of technology activity. Emissions, such as CO2 (produced from fuel combustion and reactions), CH4, N2O, F-gases, can also be accounted for in this file. See *below* for a description.

7.3.9 General features

To illustrate the data required for a generic technology in MUSE, the *electric boiler technology* is used as an example. The commodity flow for the electric boiler, capable to cover space heating and water heating energy service demands.

Fig. 1: The table below shows the basic data requirements for a typical technology, the electric boiler.

Technology: Electric Boiler ProcessName: resBoilerElectric	*Values*	*Units (input commodity unit/process activity unit)*
electricity	1.0	GWh/PJ
Output commodity	*Values*	*Unit (output commodity unit/process activity unit)*
Space cooling	0	PJ/PJ
Space heating	0.80	PJ/PJ
Water heating	0.20	PJ/PJ
Appliances	0	PJ/PJ
Lighting	0	PJ/PJ
Cooking	0	PJ/PJ
CO_2 *from reaction*	0	kt/PJ
CO_2 *from combustion*	0	kt/PJ
CO_2 *captured*	0	kt/PJ
CO_2 *stored*	0	kt/PJ
CH_4	0	kt/PJ
N_2O	0	kt/PJ
F-gases	0	kt/PJ

Below it is shown the generic structure of the input commodity file for the electric heater.

Table 8: Commodities used as consumables - Input commodities

ProcessName	RegionName	Time	Level	electricity
Unit	•	Year	•	GWh/PJ
resBoilerElectric	region1	2010	fixed	300
resBoilerElectric	region1	2030	fixed	290

ProcessName represents the technology ID and needs to be consistent across all the data inputs.

RegionName represents the region ID and needs to be consistent across all the data inputs.

Time represents the period of the simulation to which the value applies; it needs to contain at least the base year of the simulation.

Level characterises either a fixed or a flexible input type the following columns should contain the list of commodities the row.

Unit reports the unit in which the technology consumption is defined; it is for the user internal reference only.

The same structure for the csv file would also apply for the output commodity file. The input data has to be provided for the base year. Additional years within the time framework of the overall simulation can be defined. In this case, MUSE would interpolate the values between the provided periods and assume a constant value afterwards.

7.3.10 Existing Sectoral Capacity

For each technology, the decommissioning profile should be given to MUSE.

The csv file which provides the installed capacity in base year and the decommissioning profile in the future periods for each technology in a sector, in each region, should follow the structure reported in the table.

Table 9: Existing capacity of technologies: the residential boiler example

ProcessName	RegionName	Unit	2010	2020	2030	2040	2050
resBoilerElectric	region1	PJ/y	5	0.5	0	0	0
resBoilerElectric	region2	PJ/y	39	3.5	1	0.3	0

ProcessName represents the technology ID and needs to be consistent across all the data inputs.

RegionName represents the region ID and needs to be consistent across all the data inputs.

Unit reports the unit of the technology capacity; it is for the user internal reference only.

2010,. . . , 2050 represent the simulated periods.

7.3.11 Agents

In MUSE, an agent-based formulation was originally introduced for the residential and commercial building sectors []. Agents are defined using a CSV file, with one agent per row, using a format meant specifically for retrofit and new-capacity agent pairs. This CSV file can be read using `read_csv_agent_parameters()`. The data is also interpreted to some degree in the factory functions `create_retrofit_agent()` and `create_newcapa_agent()`.

For instance, we have the following CSV table:

Name	Type	AgentShare	RegionName	Objective1	SearchRule	DecisionMethod	...
A1	New	Agent5	ASEAN	EAC	all	epsilonCon	...
A4	New	Agent6	ASEAN	CapitalCosts	existing	weightedSum	...
A1	Retrofit	Agent1	ASEAN	efficiency	all	epsilonCon	...
A2	Retrofit	Agent2	ASEAN	Emissions	similar	weightedSum	...

For simplicity, not all columns are included in the example above. Though all column listed below are currently required.

The columns have the following meaning:

Name Name shared by a retrofit and new-capacity agent pair.

Type One of "New" or "Retrofit". "New" and "Retrofit" agents make up a pair with a given *name*. The demand is split into two, with one part coming from decommissioned assets, and the other coming from everything else. "Retrofit" agents invest only to make up for decommissioned assets. They are often limited in the technologies they can consider (by *SearchRule*). "New" agents invest on the rest of the demand, and can often consider more general sets of technologies.

AgentShare Name of the share of the existing capacity assigned to this agent. Only meaningful for retrofit agents. The actual share itself can be found in *Techno-data Timeslices*.

RegionName Region where an agent operates.

Objective1 First objective that an agent will try and maximize or minimize during investment. This objective should be one registered with `@register_objective`. The following objectives are available with MUSE:

- `comfort`: Comfort provided by a given technology. Comfort does not change during the simulation. It is obtained straightforwardly from *Techno-data Timeslices*.

- `efficiency`: Efficiency of the technologies. Efficiency does not change during the simulation. It is obtained straightforwardly from *Techno-data Timeslices*.
- `fixed_costs`: The fixed maintenance costs incurred by a technology. The costs are a function of the capacity required to fulfil the current demand.
- `capital_costs`: The capital cost incurred by a technology. The capital cost does not change during the simulation. It is obtained as a function of parameters found in *Techno-data Timeslices*.
- `emission_cost`: The costs associated for emissions for a technology. The costs is a function both of the amount produced (equated to the total demand in this case) and of the prices associated with each pollutant. Aliased to "emission" for simplicity.
- `fuel_consumption_cost`: Costs of the fuels for each technology, where each technology is used to fulfil the whole demand.
- `lifetime_levelized_cost_of_energy`: LCOE over the lifetime of a technology. Aliased to "LCOE" for simplicity.
- `net_present_value`: Present value of all the costs of installing and operating a technology, minus its revenues, of the course of its lifetime. Aliased to "NPV" for simplicity.
- `equivalent_annual_cost`: Annualized form of the net present value. Aliased to "EAC" for simplicity.

The weight associated with this objective can be changed using *ObjData1*. Whether the objective should be minimized or maximized depends on *Objsort1*. Multiple objectives are combined using the *DecisionMethod*

Objective2 Second objective. See *Objective1*.

Objective3: Third objective. See *Objective1*.

ObjData1 A weight associated with the *first objective*. Whether it is used will depend in large part on the *decision method*.

ObjData2 A weight associated with the *second objective*. See *ObjData1*.

ObjData3 A weight associated with the *third objective*. See *ObjData1*.

Objsort1 Whether to maximize (*True*) or minimize (*False*) the *first objective*.

Objsort2 Whether to maximize (*True*) or minimize (*False*) the *second objective*.

Objsort3 Whether to maximize (*True*) or minimize (*False*) the *third objective*.

SearchRule The search rule allows users to par down the search space of technologies to those an agent is likely to consider. The search rule is any function with a given signature, and registered with MUSE via `@register_filter`. The following search rules, defined in `filters`, are available with MUSE:

- `same_enduse`: Only allow technologies that provide the same enduse as the current set of technologies owned by the agent.
- `identity`: Allows all current technologies. E.g. disables filtering. Aliased to "all".
- `similar_technology`: Only allows technologies that have the same type as current crop of technologies in the agent, as determined by "tech_type" in *Techno-data Timeslices*. Aliased to "similar".
- `same_fuels`: Only allows technologies that consume the same fuels as the current crop of technologies in the agent. Aliased to "fueltype".
- `currently_existing_tech`: Only allows technologies that the agent already owns. Aliased to "existing".
- `currently_referenced_tech`: Only allows technologies that are currently present in the market with non-zero capacity.
- `maturity`: Only allows technologies that have achieved a given market share.

The implementation allows for combining these filters. However, the CSV data format described here does not.

DecisionMethod Decision methods reduce multiple objectives into a single scalar objective per replacement technology. They allow combining several objectives into a single metric through which replacement technologies can be ranked.

Decision methods are any function which follow a given signature and are registered via the decorator `@register_decision`. The following decision methods are available with MUSE, as implemented in `decisions`:

- `mean`: Computes the average across several objectives.
- `weighted_sum`: Computes a weighted average across several objectives.
- `lexical_comparion`: Compares objectives using a binned lexical comparison operator. Aliased to "lexo". This is a lexicographic method where objectives are compared in a specific order, for example first costs, then environmental emissions.
- `retro_lexical_comparion`: A binned lexical comparison function where the bin size is adjusted to ensure the current crop of technologies are competitive. Aliased to "retro_lexo".
- `epsilon_constraints`: A comparison method which ensures that first selects technologies following constraints on objectives 2 and higher, before actually ranking them using objective 1. Aliased to "epsilon" and "epsilon_con".
- `retro_epsilon_constraints`: A variation on epsilon constraints which ensures that the current crop of technologies are not deselected by the constraints. Aliased to "retro_epsilon".
- `single_objective`: A decision method to allow ranking via a single objective.

The functions allow for any number of objectives. However, the format described here allows only for three.

Quantity A factor used to determine the demand share of "New" agents.

MaturityThreshold Parameter for the search rule `maturity`.

7.3.12 Correlation demand files

It is possible to use macrodrivers as a way to infer the service demand. For example, one can use the expected GDP based on purchasing power parity (GDP PPP) and population in the future per region to infer the service demand using a regressor.

To do this, a minimum of three files are required:

1. A macrodrivers file
2. A file which states the regression parameters
3. A file which dictates how the demand per benchmark year is split across the timeslices.

We will go into the details of each of these files below.

Macrodrivers

An example of a shortened macrodriver file is shown below. This file contains the data for each of the years you are interested in. For example, in the file below, it contains GDP PPP in region *R1*, in the unit *millionUS$2015* for each year. It also contains the data for the population.

Table 10: Macrodrivers

Variable	RegionName	Unit	2010	2011	...
GDP|PPP	R1	millionUS$2015	1206919	1220599	...
Population	R1	million	80.0042	81.82599	...

Variable This is the variable that you would like to use in the regression for the service demand.

RegionName This is the region that the data applies to. This must correlate with the regions set in the rest of your input files, as well as the toml file.

Unit This unit can be whatever you like, however they must be consistent across all input files.

2010, 2011, ... This is the quantity of the variable per year of the simulation.

Regression Parameters

In the regression parameters file, it is necessary to state the parameters of the regression. This can be obtained from your own dataset, where you regress the service demand against GDP PPP and populaiton, for example.

An example file is shown below:

Table 11: Regression Parameters File

Sector-Name	FunctionType	Coeff	Region-Name	electric-ity	gas	heat	CO2f
Residential	logistic-sigmoid	GDPexp	R1	0	0	9.94E-02	0
Residential	logistic-sigmoid	constant	R1	0	0	0.0000434	0
Residential	logistic-sigmoid	GDPscaleLess	R1	0	0	753.1068725	0
Residential	logistic-sigmoid	GDPscale-Greater	R1	0	0	672.9316672	0

SectorName This is the sector name in which these parameters apply to.

FunctionType This is the function type you would like to MUSE to use. MUSE allows these to be:

- Exponential
- ExponentialAdj
- Logistic
- Loglog
- LogisticSigmoid
- Linear
- endogenous_demand

Your own functions can be created using the *@register_regression* hook, from the *regressions.py* file.

Coeff This is the coefficient for the respective function type. These are explicitly defined within the *regressions.py* file, as they differ between functions.

RegionName This is the region in which these parameters apply to.

Energy service (electricity, gas, heat, CO2f) Here you can specify the coefficients for the expected demand for the respective commodity.

Timeslice share

In this file, you are able to split the energy service proportionally by timeslice.

An example file is shown below:

Table 12: Timeslice share

SN	RegionName	electricity	gas	heat	CO2f	wind
1	R1	0	0	0.034835	0	0
2	R1	0	0	0.064546	0	0
3	R1	0	0	0.044569	0	0
4	R1	0	0	0.011161	0	0
5	R1	0	0	0.014145	0	0
6	R1	0	0	0.085783	0	0

SN This is the timeslice index.

RegionName This is the region in question for this data.

Energy service (electricity, gas, heat, CO2f, wind) Here you specify the proportion of each energy service for each timeslice.

7.3.13 Indices and tables

- genindex
- modindex
- search

7.4 Indices and tables

- genindex
- modindex
- search

CHAPTER

EIGHT

ADVANCED GUIDE

8.1 Extending MUSE

One key feature of the generalized sector's implementation is that it should be easy to extend. As such, MUSE can be made to run custom python functions, as long as these inputs and output of the function follow a standard specific to each step. We will look at a few here.

Below is a list of possible hooks, referenced by their implementation in the MUSE model:

- `register_interaction_net` in `muse.interactions`: a list of lists of agents that interact together.
- `register_agent_interaction` in `muse.interactions`: Given a list of interacting agents, perform the interaction.
- `register_production` in `muse.production`: A method to compute the production from a sector, given the demand and the capacity.
- `register_initial_asset_transform` in `muse.hooks`: Allows any kind of transformation to be applied to the assets of an agent, prior to investing.
- `register_final_asset_transform` in `muse.hooks`: After computing the investment, this sets the assets that will be owned by the agents.
- `register_demand_share` in `muse.demand_share`: During agent investment, this is the share of the demand that an agent will try and satisfy.
- `register_filter` in `muse.filters`: A filter to remove technologies from consideration, during agent investment.
- `register_objective` in `muse.objectives`: A quantity which allows an agent to compare technologies during investment.
- `register_decision` in `muse.decisions`: A transformation applied to aggregate multiple objectives into a single objective during agent investment, e.g. via a weighted sum.
- `register_investment` in `muse.investment`: During agent investment, matches the demand for future investment using the decision metric above.
- `register_output_quantity` in `muse.output.sector`: A sectorial quantity to output for postmortem analysis.
- `register_output_sink` in `muse.outputs`: A *place* to store an output quantity, e.g. a file with a given format, a database on premise or on the cloud, etc…
- `register_carbon_budget_fitter` in `muse.carbon_budget`
- `register_carbon_budget_method` in `muse.carbon_budget`
- `register_sector`: Registers a function that can create a sector from a muse configuration object.

8.1.1 Extending outputs

MUSE can be used to save custom quantities as well as data for analysis. There are two steps to this process:

- Computing the quantity of interest
- Store the quantity of interest in a sink

In practice, this means that we can compute any quantity, such as capacity or consumption of an energy source and save it to a csv file, or a netcdf file.

Output extension

To demonstrate this, we will compute a new edited quantity of consumption, then save it as a text file.

The current implementation of the quantity of consumption found in `muse.outputs.sector` filters out values of 0. In this example, we would like to maintain the values of 0, but do not want to edit the source code of MUSE.

This is rather simple to do using MUSE's hooks.

First we create a new function called `consumption_zero` as follows:

[1]:
```
from muse.outputs.sector import register_output_quantity
from muse.outputs.sector import market_quantity
from xarray import Dataset, DataArray
from typing import Optional, List, Text

@register_output_quantity
def consumption_zero(
    market: Dataset,
    capacity: DataArray,
    technologies: Dataset,
):
    """Current consumption."""
    result = (
        market_quantity(market.consumption, sum_over="timeslice", drop=None)
        .rename("consumption")
        .to_dataframe()
        .round(4)
    )
    return result
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_204/4248002360.py in <module>
----> 1 from muse.outputs.sector import register_output_quantity
      2 from muse.outputs.sector import market_quantity
      3 from xarray import Dataset, DataArray
      4 from typing import Optional, List, Text
      5

ModuleNotFoundError: No module named 'muse'
```

The function we created takes three arguments. These arguments (`market`, `capacity` and `technology`) are mandatory for the `@register_output_quantity` hook. Other hooks require different arguments.

Whilst this function is very similar to the `consumption` function in `muse.outputs.sector`, we have modified it slightly by allowing for values of `0`.

The important part of this function is the `@register_output_quantity` decorator. This decorator ensures that this new quantity is addressable in the TOML file. Notice that we did not need to edit the source code to create our new function.

Next, we can create a sink to save the output quantity previously registered. For this example, this sink will simply dump the quantity it is given to a file, with the "Hello world!" message:

[2]:
```
from typing import Any, Text
from muse.outputs.sinks import register_output_sink, sink_to_file

@register_output_sink(name="txt")
@sink_to_file(".txt")
def text_dump(data: Any, filename: Text) -> None:
    from pathlib import Path
    Path(filename).write_text(f"Hello world!\n\n{data}")
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_204/31753769.py in <module>
      1 from typing import Any, Text
----> 2 from muse.outputs.sinks import register_output_sink, sink_to_file
      3
      4 @register_output_sink(name="txt")
      5 @sink_to_file(".txt")

ModuleNotFoundError: No module named 'muse'
```

The code above makes use of two dectorators: `@register_output_sink` and `@sink_to_file`.

`@register_output_sink` registers the function with MUSE, so that the sink is addressable from a TOML file. The second one, `@sink_to_file`, is optional. This adds some nice-to-have features to sinks that are files. For example, a way to specify filenames and check that files cannot be overwritten, unless explicitly allowed to.

Next, we want to modify the TOML file to actually use this output type. To do this, we add a section to the output table:

```
[[sectors.residential.outputs]]
quantity = "consumption_zero"
sink = "txt"
filename = "{cwd}/{default_output_dir}/{Sector}{Quantity}{year}{suffix}"
```

The last line above allows us to specify the name of the file. We could also use `sector` above or `quantity`.

There can be as many sections of this kind as we like in the TOML file, which allow for multiple outputs.

Next, we first copy the default model provided with muse to a local subfolder called "model". Then we read the `settings.toml` file and modify it using python. You may prefer to modify the `settings.toml` file using your favorite text editor. However, modifying the file programmatically allows us to routinely run this notebook as part of MUSE's test suite and check that the tutorial it is still up to date.

[3]:
```
from pathlib import Path
from toml import load, dump
from muse import examples

model_path = examples.copy_model(overwrite=True)
settings = load(model_path / "settings.toml")
new_output = {
    "quantity": "consumption_zero",
```

(continues on next page)

(continued from previous page)

```
    "sink":  "txt",
    "overwrite": True,
    "filename": "{cwd}/{default_output_dir}/{Sector}{Quantity}{year}{suffix}",
}
settings["sectors"]["residential"]["outputs"].append(new_output)
dump(settings, (model_path / "modified_settings.toml").open("w"))
settings
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_204/267986346.py in <module>
      1 from pathlib import Path
----> 2 from toml import load, dump
      3 from muse import examples
      4
      5 model_path = examples.copy_model(overwrite=True)

ModuleNotFoundError: No module named 'toml'
```

We can now run the simulation. There are two ways to do this. From the command-line, where we can do:

```
python3 -m muse data/commercial/modified_settings.toml
```

(note that slashes may be the other way on Windows). Or directly from the notebook:

[4]:
```
import logging
from muse.mca import MCA
logging.getLogger("muse").setLevel(0)
mca = MCA.factory(model_path / "modified_settings.toml")
mca.run();
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_204/1115410891.py in <module>
      1 import logging
----> 2 from muse.mca import MCA
      3 logging.getLogger("muse").setLevel(0)
      4 mca = MCA.factory(model_path / "modified_settings.toml")
      5 mca.run();

ModuleNotFoundError: No module named 'muse'
```

We can now check that the simulation has created the files that we expect. We also check that our "Hello, world!" message has printed:

[5]:
```
all_txt_files = sorted((Path() / "Results").glob("Residential*.txt"))
assert "Hello world!" in all_txt_files[0].read_text()
all_txt_files
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_204/1678409945.py in <module>
      1 all_txt_files = sorted((Path() / "Results").glob("Residential*.txt"))
----> 2 assert "Hello world!" in all_txt_files[0].read_text()
```

(continues on next page)

(continued from previous page)

```
      3 all_txt_files

AssertionError:
```

Our model output the files we were expecting and passed the `assert` statement, meaning that it could find the "Hello world!" messages in the outputs.

8.1.2 Adding TOML parameters to the outputs

It would be useful if we could pass parameters from the TOML file to our new functions `consumption_zero` and `text_dump`. For example, in our previous iteration the consumption output was aggregating the data by `"timeslice"`, by hardcoding the variable. We can pass a parameter which could do this by setting the `sum_over` parameter to be `True`. In addition, we could change the message output by a new `text_dump` function.

Not all hooks are this flexible (for historical reasons, rather than any intrinsic difficulty). However, for outputs, we can do this as follows:

[6]:
```
@register_output_quantity(overwrite=True)
def consumption_zero(
    market: Dataset,
    capacity: DataArray,
    technologies: Dataset,
    sum_over: Optional[List[Text]] = None,
    drop: Optional[List[Text]] = None,
    rounding: int = 4,
):
    """Current consumption."""
    result = (
        market_quantity(market.consumption, sum_over=sum_over, drop=drop)
        .rename("consumption")
        .to_dataframe()
        .round(rounding)
    )
    return result


@register_output_sink(name="txt", overwrite=True)
@sink_to_file(".txt")
def text_dump(
    data: Any,
    filename: Text,
    msg : Optional[Text] = "Hello, world!"
) -> None:
    from pathlib import Path
    Path(filename).write_text(f"{msg}\n\n{data}")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_204/717001874.py in <module>
----> 1 @register_output_quantity(overwrite=True)
      2 def consumption_zero(
      3     market: Dataset,
```

(continues on next page)

(continued from previous page)

```
      4     capacity: DataArray,
      5     technologies: Dataset,

NameError: name 'register_output_quantity' is not defined
```

We simply added parameters as arguments to both of our functions: `consumption_zero` and `text_dump`.

Note: The overwrite argument allows us to overwrite previously defined registered functions. This is useful in a notebook such as this. But it should not be used in general. If overwrite were false, then the code would issue a warning and it would leave the TOML to refer to the original functions at the beginning of the notebook. This is useful when using custom modules.

Now we can modify the output section to take additional arguments:

```
[[sectors.commercial.outputs]]
quantity.name = "consumption_zero"
quantity.sum_over = "timeslice"
sink.name = "txt"
sink.filename = "{cwd}/{default_output_dir}/{Sector}{Quantity}{year}{suffix}"
sink.msg = "Hello, you!"
sink.overwrite = True
```

Here, we still want to use the `consumption_zero` function and the `txt` sink. But we would like to change the message from "Hello world!" to "Hello you!" within the `TOML` file.

Now, both sink and quantity are dictionaries which can take any number of arguments. Previously, we were using a shorthand for convenience. Again, we create a new settings file, and run this with our new parameters, which interface with our new functions.

[7]:
```
from pathlib import Path
from toml import load, dump
from muse import examples

model_path = examples.copy_model(overwrite=True)
settings = load(model_path / "settings.toml")
settings["sectors"]["residential"]["outputs"] = [
    {
        "quantity":{
            "name": "consumption_zero",
            "sum_over": "timeslice"
        },
        "sink":{
            "name": "txt",
            "filename": "{cwd}/{default_output_dir}/{Sector}{Quantity}{year}{suffix}",
            "msg": "Hello, you!",
            "overwrite": True,
        }

    }
]

dump(settings, (model_path / "modified_settings_2.toml").open("w"))
settings
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_204/4294915811.py in <module>
      1 from pathlib import Path
----> 2 from toml import load, dump
      3 from muse import examples
      4
      5 model_path = examples.copy_model(overwrite=True)

ModuleNotFoundError: No module named 'toml'
```

We then run the simulation again:

[8]:
```
mca = MCA.factory(model_path / "modified_settings_2.toml")
mca.run();
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_204/267455335.py in <module>
----> 1 mca = MCA.factory(model_path / "modified_settings_2.toml")
      2 mca.run();

NameError: name 'MCA' is not defined
```

And we can check the parameters were used accordingly:

[9]:
```
all_txt_files = sorted((Path() / "Results").glob("Residential*.txt"))
assert len(all_txt_files) == 7
assert "Hello, you!" in all_txt_files[0].read_text()
all_txt_files
```

[9]:
```
[PosixPath('Results/ResidentialConsumption_Zero2020.txt'),
 PosixPath('Results/ResidentialConsumption_Zero2025.txt'),
 PosixPath('Results/ResidentialConsumption_Zero2030.txt'),
 PosixPath('Results/ResidentialConsumption_Zero2035.txt'),
 PosixPath('Results/ResidentialConsumption_Zero2040.txt'),
 PosixPath('Results/ResidentialConsumption_Zero2045.txt'),
 PosixPath('Results/ResidentialConsumption_Zero2050.txt')]
```

Again, we can see that the number of output files generated were as we expected and that our new message "Hello, you!" was found within these files. This means that our output and sink functions worked as expected.

8.1.3 Where to store new functionality

As previously demonstrated, we can easily add new functionality to MUSE. However, running a jupyter notebook is not always the best approach. It is also possible to store functions in an arbitrary pthon file, such as the following:

[10]:
```
%%writefile mynewfunctions.py
from typing import Any, Text
from muse.outputs.sinks import register_output_sink, sink_to_file

@register_output_sink(name="txt")
@sink_to_file(".txt")
```

(continues on next page)

(continued from previous page)

```
def text_dump(data: Any, filename: Text) -> None:
    from pathlib import Path
    Path(filename).write_text(f"Hello world!\n\n{data}")
```

```
Overwriting mynewfunctions.py
```

We can then tell the TOML file where to find it:

```
plugins = "{cwd}/mynewfunctions.py"

[[sectors.commercial.outputs]]
quantity = "capacity"
sink = "dummy"
overwrite = true
```

Alternatively, `plugin` can also be given a list of paths rather than just a single one, as done below.

[11]:
```
settings = load(model_path / "settings.toml")
settings["plugins"] = ["{cwd}/mynewfunctions.py"]
settings["sectors"]["residential"]["outputs"] = [
    {
        "quantity": "capacity",
        "sink":  "dummy",
        "overwrite": "true"
    }
]
dump(settings, (model_path / "modified_settings.toml").open("w"))
settings
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_204/3840354473.py in <module>
----> 1 settings = load(model_path / "settings.toml")
      2 settings["plugins"] = ["{cwd}/mynewfunctions.py"]
      3 settings["sectors"]["residential"]["outputs"] = [
      4     {
      5         "quantity": "capacity",

NameError: name 'load' is not defined
```

8.1.4 Next steps

In the next section we will output a technology filter, to stop agents from investing in a certain technology, and a new metric to combine multiple objectives.

8.2 Further extending MUSE

8.2.1 Adding a search space filter

[1]:
```
from xarray import Dataset, DataArray

from muse.agents import Agent
from muse.filters import register_filter

@register_filter
def no_ccgt_filter(
    agent: Agent,
    search_space: DataArray,
    technologies: Dataset,
    market: Dataset
) -> DataArray:
    """Excludes gasCCGT."""
    dropped_tech = search_space.where(search_space.replacement != "windturbine")
    return search_space & search_space.replacement.isin(dropped_tech.replacement)
```

[2]:
```
import logging
from muse.mca import MCA
from muse import examples

# model_path = examples.copy_model(overwrite=True)
logging.getLogger("muse").setLevel(0)
mca = MCA.factory("model/settings.toml")
mca.run();
```

```
-- 2020-11-26 16:43:53 - muse.sectors.register - INFO
Sector legacy registered.

-- 2020-11-26 16:43:53 - muse.sectors.register - INFO
Sector preset registered, with alias presets.

-- 2020-11-26 16:43:53 - muse.sectors.register - INFO
Sector default registered.

-- 2020-11-26 16:43:58 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:03 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:08 - muse.mca - WARNING
Check growth constraints for wind.
```

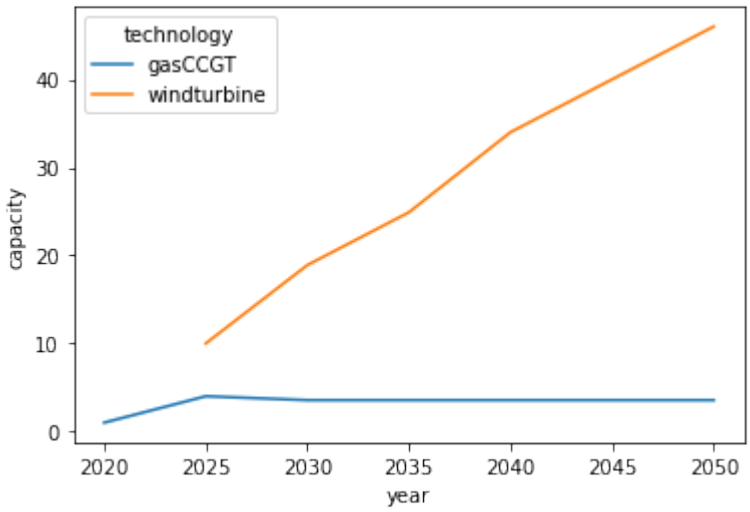
(continues on next page)

(continued from previous page)

```
-- 2020-11-26 16:44:12 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:17 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:21 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:25 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:30 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:34 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:38 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:43 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:47 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:44:51 - muse.mca - WARNING
Check growth constraints for wind.
```

[3]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot

results = pd.read_csv("Results/MCACapacity.csv")
results
sns.lineplot(data=results[results.sector=="power"], x='year', y='capacity', hue=
↪'technology')
```

[3]:
```
<AxesSubplot:xlabel='year', ylabel='capacity'>
```



8.2.2 Registering a custom decision function

Next, we would like to add an additional decision function. A decision function is a transformation applied to aggregate multiple objectives into a single objective during agent investment. For example, through the use of a weighted sum.

In this example, we would like to take the median objective. However, the functions predefined in MUSE don't include this functionality. MUSE contains examples such as the mean, weighted sum and a single objective. Therefore we will have to register and create our own.

Now, we create our new `median_objective` function:

[4]:
```
from muse.decisions import register_decision
from typing import Any
from xarray import Dataset

@register_decision
def median_objective(
    objectives: Dataset,
    parameters: Any,
    **kwargs
) -> DataArray:
    from xarray import concat
    allobjectives = concat(objectives.data_vars.values(), dim="concat_var")
    return allobjectives.median(set(allobjectives.dims) - {"asset", "replacement"})
```

After importing the decorator function, `register_decision`, and ensuring that we decorate our new function with `@register_decision`, we are able to create our new function as above.

Our new function `median_objective` modifies the `mean` function already built into MUSE, with one difference. Replacing the return value, from `allobjectives.mean` to `allobjectives.mean`.

```
@register_decision
def mean(objectives: Dataset, *args, **kwargs) -> DataArray:
    """Mean over objectives."""
    from xarray import concat
```

(continues on next page)

(continued from previous page)

```
    allobjectives = concat(objectives.data_vars.values(), dim="concat_var")
    return allobjectives.mean(set(allobjectives.dims) - {"asset", "replacement"})
```

Of course, you are free to make your functions as complicated as you like, depending on your own requirements.

Next, we must edit our `Agents.csv` file. We will modify the `default` example for this tutorial. We change the first two entry rows, to be as follows:

```
Agent1,A1,1,R1,LCOE,NPV,EAC,1,,,FALSE,,,all,median_objective,1,-1,inf,New
Agent2,A1,2,R1,LCOE,NPV,EAC,1,,,FALSE,,,all,median_objective,1,-1,inf,Retrofit
```

Here, we add the NPV and EAC decision metrics, as well as replacing the `singleObj DecisionMethod` to `median_objective`.

Now we are able to run our modified model as before:

[5]:
```
logging.getLogger("muse").setLevel(0)
mca = MCA.factory("../tutorial-code/new-decision-metric/settings.toml")
mca.run();
```

```
-- 2020-11-26 16:44:59 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:02 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:04 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:09 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:13 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:17 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:20 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:25 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:28 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:32 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:35 - muse.mca - WARNING
Check growth constraints for wind.

-- 2020-11-26 16:45:39 - muse.mca - WARNING
```

(continues on next page)

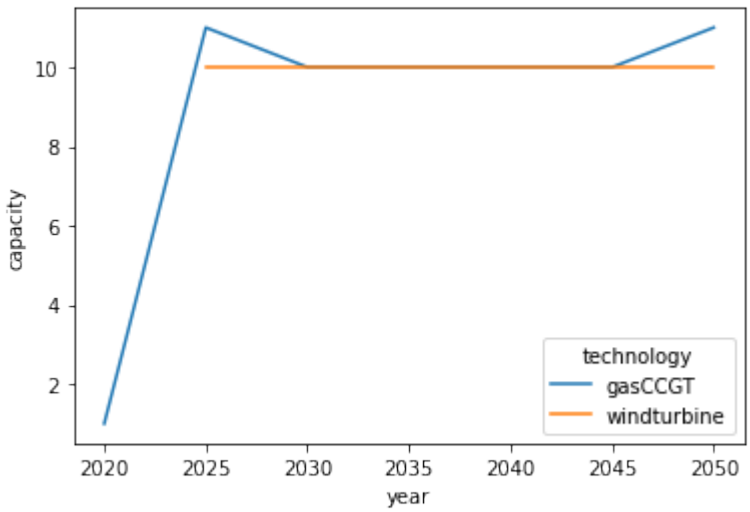
(continued from previous page)

```
Check growth constraints for wind.

-- 2020-11-26 16:45:43 - muse.mca - WARNING
Check growth constraints for wind.
```

Again, we visualise the power sector:

[6]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot

results = pd.read_csv("Results/MCACapacity.csv")
results
sns.lineplot(data=results[results.sector=="power"], x='year', y='capacity', hue=
↪'technology')
```

[6]:
```
<AxesSubplot:xlabel='year', ylabel='capacity'>
```



We see a different scenario emerge through these different decision metrics. This shows the importance of decision metrics when making long-term investment decisions.

8.2.3 End of tutorials

In these tutorials you have seen the ways in which you can modify MUSE. All of these methods can be combined and extended to make as simple or complex model as you wish. Feel free to experiment and come up with your own ideas for your future work!

For further information, we refer you to the API in the next section.

8.3 API

8.3.1 Market Clearing Algorithm

Main MCA

Carbon Budget

8.3.2 Sectors and associated functionality

AbstractSector

Sector

Subsector

PresetSector

LegacySector

Production

Agent Interactions

8.3.3 Agents and associated functionalities

Objectives

Search Rules

Decision Methods

Investment Methods

Demand Share

Constraints:

Initial and Final Asset Transforms

8.3.4 Reading the inputs

8.3.5 Writing Outputs

Sinks

Sectorial Outputs

8.3.6 Quantities

8.3.7 Demand Matching Algorithm

8.3.8 Miscellaneous

Timeslices

Commodities

Regression functions

Functionality Registration

Utilities

Examples

CHAPTER

NINE

FAQS

9.1 I get a demand matching error

If you get an error, similar to the following:

```
muse/src/muse/investments.py", line 328, in scipy_match_demand
    raise LinearProblemError("LP system could not be solved", res)
muse.investments.LinearProblemError: ('LP system could not be solved',      con:␣
↪array([], dtype=float64)
    fun: 15053.021584161033
message: 'The algorithm terminated successfully and determined that the problem is␣
↪infeasible.'
    nit: 11
slack: array([ 0.12188998,  2.27624361,  1.72261721,  0.65248747, -0.22047042,
        0.62430284,  5.80377518,  1.96572729,  1.5198397 ,  0.61823037,
    -0.21871122,  0.59457721,  2.49343834,  1.10418509,  0.84691291,
        0.3260397 , -0.2204754 ,  0.31897793,  4.36788165,  4.9152109 ,
        6.91218128, 23.93103992,  8.76641572,  7.35194341,  3.24577114,
    -0.26032861,  2.8121707 ])
status: 2
success: False
    x: array([32.03211835, 35.0847891 , 43.08781872, 11.8901544 ,  3.72977858,
        3.08220055,  1.74992141,  1.42167486,  1.77810604,  7.35302074,
        4.61267067,  3.74287866,  2.01312881,  1.53439081,  2.03678197,
        4.68786478,  2.48646647,  2.02560833,  1.11022092,  0.93860572,
        1.11728269]))
```

this is because the optimisation algorithm can not find a solution to match supply with demand. This is often because the constraints placed on the technologies do not allow for high enough growth to meet a growing demand.

A solution to this is to increase the limits of technologies in the relevant *Technodata.csv*. For example, by increasing the *MaxCapacityAddition*, *MaxCapacityGrowth* and/or *TotalCapacityLimit* variables for the respective technologies.

9.2 What units should I be using within MUSE?

The units within MUSE should be consistent. Therefore it is up to you which units you use. You could use, like the examples, petajoules (PJ), however, the units used must be the same across each of the sectors, and each of the input files. MUSE does not make any unit conversion internally.

9.3 How do I activate my conda environment?

To activate your conda environment, run the following command:

```
conda activate <name-of-environment>
```

For example, if you want to activate your conda environment called muse run:

```
conda activate muse
```

9.4 How do I know my conda environment is activated?

Your conda environment is activated if you see something similar to the following in your Anaconda Powershell Prompt or command line:

```
(muse) PS C:/Users/<my-username>
```

9.5 I get a "Cannot find command 'git' - do you have 'git' installed in your PATH?" error

This is because you do not have git installed in your conda environment. To resolve this, run:

```
conda install git
```

9.6 When I input my GitHub password into Anaconda Powershell Prompt to download MUSE, I don't see any input

This is normal behaviour. It is done to stop people watching as you type your password over your shoulder. Just continue typing in your password as you would on a website.

CHAPTER

TEN

API

10.1 Market Clearing Algorithm

10.1.1 Main MCA

10.1.2 Carbon Budget

10.2 Sectors and associated functionality

10.2.1 AbstractSector

10.2.2 Sector

10.2.3 Subsector

10.2.4 PresetSector

10.2.5 LegacySector

10.2.6 Production

10.2.7 Agent Interactions

10.3 Agents and associated functionalities

10.3.1 Objectives

10.3.2 Search Rules

10.3.3 Decision Methods

10.3.4 Investment Methods

10.3.5 Demand Share

10.3.6 Constraints:

10.3.7 Initial and Final Asset Transforms

10.4 Reading the inputs

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

Anaconda, 1
Anaconda Prompt, 1

B

Benchmark years, 1

E

End-use demand, 1
End-use technology, 1
Endogenous quantity, 1
Energy service, 1
Energy system, 1
Equilibrium, 1
Exogenous quantity, 1

I

Imperfect information, 1

L

Levelised cost of electricity, 1
Limited foresight, 1

O

Open-source, 1

P

Pandas, 1
Petajoules (*PJ*), 2

S

Scenario analysis, 2
Seaborn, 2
Simulation, 2

T

Timeslice, 2