nempy

Release 0.0.1

Nick Gorman

Dec 11, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Nempy is an open-source python package that can be used to model the dispatch procedure of the Australian National Electricity Market (NEM). The dispatch process is at the core of many market modelling studies. Nempy allows users to easily configure a dispatch model to fit the relevant research question. Furthermore, if extra functionality is needed, the python implementation and open-source licencings allow the user to make modifications. Nempy is feature rich, flexible, can recreate historical dispatch with a high degree of accuracy, runs fast, and has detailed documentation.

The Nempy source code is on GitHub: https://github.com/UNSW-CEEM/nempy.

A brief introduction to the NEM can be found here: https://aemo.com.au/-/media/Files/Electricity/NEM/National-Electricity-Market-Fact-Sheet.pdf

1.1 Author

Nempy's development is being led by Nick Gorman as part of his PhD candidature at the Collaboration on Energy and Environmental Markets at the University of New South Wales' School of Photovoltaics and Renewable Energy Engineering. (https://www.ceem.unsw.edu.au/).

1.2 Support

You can seek support for using Nempy using the discussion tab on GitHub (https://github.com/UNSW-CEEM/nempy/discussions), checking the issues register (https://github.com/UNSW-CEEM/nempy/issues), or by contacting Nick directly (n.gorman at unsw.edu.au).

1.3 Future support and maintenance

Planning to continue support and maintenance for Nempy after the PhD project is complete is currently underway. If Nempy is useful to your work, research, or business, please reach out and inform us so we can consider your use case and needs.

1.4 Example use cases

Nempy is intended for analysts and modellers studying the NEM either in industry or academic. It can be used either as is, or as building block in a large modelling tool. Some potential use case are:

1. As a tool for studying the dispatch process itself. The example shown in the *section on model accuracy* below demonstrates how model simplifications effects accuracy, this is potentially useful information for other NEM modellers either using Nempy or other modelling tools.
2. As a building block in agent based market models, as part of the environment for agents to interact with.
3. To answer counter factual questions about historical dispatch outcomes. For example, how removing a network constraint would have effected dispatch and pricing outcomes?
4. As a reference implementation of the NEM's dispatch procedure. Published documentation can lack detail, studying the source code of Nempy may be useful for some NEM analysts to gain a better understanding of the dispatch procedure.

1.5 Dispatch Procedure Outline

The main task of the dispatch procedure is the construction and solving of a mixed integer linear problem (MIP) to find the least cost set of dispatch levels for generators and scheduled loads. Note, in this optimisation the dispatch of scheduled loads is treated as a negative cost, this makes the least cost optimisation equivalent to maximising the value of market trade. The construction of the MIP as implemented by Nempy proceeds roughly as follows:

1. Bids from generators and loads are preprocessed, some FCAS bids are excluded if they do not meet a set of inclusion criteria set out by AEMO (`FCAS Model in NEMDE`).
2. For each bid a decision variable in the MIP is created, the cost of the variable in the objective function is the bid price, and the price is adjusted by a loss factor if one is provided.
3. For each market region a constraint forcing generation to equal demand is created.
4. The rest of the market features are implemented as additional variables and/or constraints in the MIP, for example:
   - unit ramp rates are converted to a set MW ramp that units can achieve over the dispatch interval, and the sum of a unit's dispatch is limited by this MW value
   - interconnectors are formulated as additional decision variables that link the supply equals demand constraints of the interconnected regions, and are combined with constraints sets that enforce interconnector losses as a function of power flow
5. The MIP is solved to determined interconnector flows and dispatch targets, the MIP is then converted to a linear problem, and re-solved, such that market prices can be determined from constraint shadow prices.

Differences between Nempy and the dispatch procedure:

1. While updated functionality in Nempy 2.0.0 now provides the capability to calculate RHS values dynamically based on SCADA and other data sources, the detailed examples provided for recreating dispatch only calculate RHS values relating to the Basslink switch, and other RHS values are taken from the NEMDE solution file.

1.6 Features

- **Energy bids**: between one and ten price quantity bid pairs can be provided for each generator or load bidding in the energy market
- **Loss factors**: loss factors can be provided for each generator and load
- **FCAS bids**: between one and ten price quantity bid pairs can be provided for each generator or load bidding in each of the eight FCAS markets
- **Ramp rates**: unit ramp rates can be set
- **FCAS trapezium constraints**: a set of trapezium constraints can be provided for each FCAS bid, these ensure FCAS is co-optimised with energy dispatch and would be technically deliverable
- **Fast start dispatch inflexibility profiles**: dispatch inflexibility profiles can be provided for unit commitment of fast-start plants
- **Interconnectors and losses**: interconnectors between each market region can be defined, non-linear loss functions and interpolation breakpoints for their linearisation can be provided
- **Generic constraints**: generic constraints that link across unit output, FCAS enablement and interconnector flows can be defined
- **Elastic constraints**: constraints can be made elastic, i.e. a violation cost can be set for constraints
- **Tie-break constraints**: constraints that minimise the difference in dispatch between energy bids for the same price can be enabled
- **Market clearing prices**: market prices are returned for both energy and FCAS markets, based on market constraint shadow prices
- **Historical inputs**: tools for downloading dispatch inputs from AEMO's NEMWeb portal and preprocessing them for compatibility with the nempy SpotMarket class are available
- **Input validation**: optionally check user inputs and raise descriptive errors when they do not meet the expected criteria
- **Adjustable dispatch interval**: a dispatch interval of any length can be used

1.7 Flexibility

Nempy is designed to have a high degree of flexibility, it can be used to implement very simple merit order dispatch models, highly detailed models that seek to re-create the real world dispatch procedure, or a model at the many levels of intermediate complexity. A set of *examples,* demonstrating this flexibility are available. Most inputs are passed to nempy as pandas DataFrame objects, which means Nempy can easily source inputs from other python code, SQL databases, CSVs and other formats supported by the pandas' interface.

1.8 Accuracy

The accuracy with which Nempy represents the NEM's dispatch process can be measured by re-creating historical dispatch results. This is done for a given dispatch interval by downloading the relevant historical inputs such as unit initial operating levels, bids and generic constraints, processing these inputs so they are compatible with the Nempy SpotMarket class, and finally dispatching the spot market. The results can then be compared to historical results to gauge the model's accuracy. Figure 1 shows the results of this process for 1000 randomly selected dispatch intervals in 2019, comparing the modelled NSW energy price with historical prices. Here the model is configured to maximally reflect the NEM's dispatch procedure (not including the Basslink switch run). The code to produce the results shown in this figure is available here. Figure 2 shows a similar comparison, but without FCAS markets or generic constraints. The code to produce the results shown in Figure 2 is available here. The simpler model produces a similar number of medianly priced intervals, however, outcomes for extreme ends of the price duration curve differ significantly from historical values.

Figure 1: A comparison of the historical NSW reference node price, prior to scaling or capping, with the price calculated using nempy. The nempy model was configured to maximally replicated the NEM dispatch process and 1000 randomly selected intervals were used.

Figure 2: A comparison of the historical NSW reference node price, prior to scaling or capping, with the price calculated using Nempy. The Nempy model was configured without FCAS markets or generic constraints and 1000 randomly selected intervals were used.

1.9 Run-time

The run-time for Nempy to calculate dispatch depends on several factors, the complexity of the model implemented, time taken to load inputs, the mixed-integer linear solver used and of course the hardware. Run-times reported here used an Intel® Xeon(R) W-2145 CPU @ 3.70 GHz. For the model results shown in Figure 1, including time taken to load inputs from the disk and using the open-source solver CBC, the average run-time per dispatch interval was 2.54 s. When the proprietary solver Gurobi was used, a run-time of 1.84 s was achieved. For the results shown in Figure 2, the run-times with CBC and Gurobi were 1.02 s and 0.98 s respectively, indicating that for simpler models the solver used has a smaller impact on run-time. For the simpler model, the time to load inputs is increased significantly by the loading of historical NEMDE input/output XML files which takes approximately 0.4 s. Importantly, this means it will be possible to speed up simpler models by sourcing inputs from different data storage formats.

Notes:

- Information on solvers is provided is provided in the reference documentation of the SpotMarket class.
- The total runtime was calculated using the python time module and measuring the time taken from the loading of inputs to the extraction of results from the model. The runtime of different sub-process, i.e. loading of the XML file, was measured by inserting timing code into the Nempy source code where required.

1.10 Documentation

Nempy has a detailed set of documentation, mainly comprising of two types: examples and reference documentation. The examples aim to show how Nempy can be used and how it works in a practical manner. A number of simple examples focus on demonstrating the use of subsets of the package's features in isolation in order to make them easier to understand. The more complex examples show how features can be combined to build models more suitable for analysis. The reference documentation aims to cover all the package's public APIs (the classes, methods and functions accessible to the user), describing their use, inputs, outputs and any side effects.

1.11 Ongoing work

Enhancements:

- The 1 second raise and lower contingency FCAS markets are in process of being added to Nempy.

1.12 Dependencies

- pandas >=1.0.0, <2.0.0
- mip>=1.11.0, <2.0.0: https://github.com/coin-or/python-mip)
- xmltodict==0.12.0: https://github.com/martinblech/xmltodict)
- requests>=2.0.0, <3.0.0

CHAPTER TWO

INSTALLATION

Installing nempy to use in your project is easy.

pip install nempy

To install for development purposes, such as adding new features. Download the source code, unzip, cd into the directory, then install.

pip install e .[dev]

Then the test suite can be run using.

python -m pytest

CHAPTER THREE

EXAMPLES

A number of examples of how to use Nempy are provided below. Examples 1 to 5 are simple and aim introduce various market features that can be modelled with Nempy in an easy to understand way, the dispatch and pricing outcomes are explained in inline comments where the results are printed. Examples 6 and 7 show how to use the historical data input preparation tools provided with Nempy to recreate historical dispatch intervals. Historical dispatch and pricing outcomes can be difficult to interpret as they are usually the result of complex interactions between the many features of the dispatch process, for these example the results are plotted in comparison to historical price outcomes. Example 8 demonstrates how the outputs of one dispatch interval can be used as the initial conditions of the next dispatch interval to create a time sequential model, additionally the current limitations with the approach are briefly discussed.

3.1 1. Bid stack equivalent market

This example implements a one region bid stack model of an electricity market. Under the bid stack model, generators are dispatched according to their bid prices, from cheapest to most expensive, until all demand is satisfied. No loss factors, ramping constraints or other factors are considered.

```python
import pandas as pd
from nempy import markets

# Volume of each bid, number of bands must equal number of bands in price_bids.
volume_bids = pd.DataFrame({
    'unit': ['A', 'B'],
    '1': [20.0, 50.0],  # MW
    '2': [20.0, 30.0],  # MW
    '3': [5.0, 10.0]  # More bid bands could be added.
})

# Price of each bid, bids must be monotonically increasing.
price_bids = pd.DataFrame({
    'unit': ['A', 'B'],
    '1': [50.0, 50.0],  # $/MW
    '2': [60.0, 55.0],  # $/MW
    '3': [100.0, 80.0]  # . . .
})

# Other unit properties
unit_info = pd.DataFrame({
    'unit': ['A', 'B'],
    'region': ['NSW', 'NSW'],  # MW
})
```

(continues on next page)

(continued from previous page)

```

# The demand in the region\s being dispatched
demand = pd.DataFrame({
    'region': ['NSW'],
    'demand': [115.0]  # MW
})

# Create the market model
market = markets.SpotMarket(unit_info=unit_info, market_regions=['NSW'])
market.set_unit_volume_bids(volume_bids)
market.set_unit_price_bids(price_bids)
market.set_demand_constraints(demand)

# Calculate dispatch and pricing
market.dispatch()

# Return the total dispatch of each unit in MW.
print(market.get_unit_dispatch())
#   unit service  dispatch
# 0    A  energy      40.0
# 1    B  energy      75.0

# Understanding the dispatch results: Unit A's first bid is 20 MW at 50 $/MW,
# and unit B's first bid is 50 MW at 50 $/MW, as demand for electricity is
# 115 MW both these bids are need to meet demand and so both will be fully
# dispatched. The next cheapest bid is 30 MW at 55 $/MW from unit B, combining
# this with the first two bids we get 100 MW of generation, so all of this bid
# will be dispatched. The next cheapest bid is 20 MW at 60 $/MW from unit A, by
# dispatching 15 MW of this bid we get a total of 115 MW generation, and supply
# meets demand so no more bids need to be dispatched. Adding up the dispatched
# bids from each generator we can see that unit A will be dispatch for 40 MW
# and unit B will be dispatch for 75 MW, as given by our bid stack market model.

# Return the price of energy in each region.
print(market.get_energy_prices())
#   region  price
# 0    NSW   60.0

# Understanding the pricing result: In this case the marginal bid, the bid
# that would be dispatch if demand increased is the 60 $/MW bid from unit
# B, thus this bid sets the price.

# Additional Detail: The above is a simplified interpretation
# of the pricing result, note that the price is actually taken from the
# underlying linear problem's shadow price for the supply equals demand constraint.
# The way the problem is formulated if supply sits exactly between two bids,
# for example at 120.0 MW, then the price is set by the lower rather
# than the higher bid. Note, in practical use cases if the demand is a floating point
# number this situation is unlikely to occur.
```

3.2 2. Unit loss factors, capacities and ramp rates

A simple example with two units in a one region market, units are given loss factors, capacity values and ramp rates. The effects of loss factors on dispatch and market prices are explained.

```
import pandas as pd
from nempy import markets

# Volume of each bid, number of bands must equal number of bands in price_bids.
volume_bids = pd.DataFrame({
    'unit': ['A', 'B'],
    '1': [20.0, 50.0],  # MW
    '2': [25.0, 30.0],  # MW
    '3': [5.0, 10.0]  # More bid bands could be added.
})

# Price of each bid, bids must be monotonically increasing.
price_bids = pd.DataFrame({
    'unit': ['A', 'B'],
    '1': [40.0, 50.0],  # $/MW
    '2': [60.0, 55.0],  # $/MW
    '3': [100.0, 80.0]  # . . .
})

# Factors limiting unit output.
unit_limits = pd.DataFrame({
    'unit': ['A', 'B'],
    'initial_output': [0.0, 0.0],  # MW
    'capacity': [55.0, 90.0],  # MW
    'ramp_up_rate': [600.0, 720.0],  # MW/h
    'ramp_down_rate': [600.0, 720.0]  # MW/h
})

# Other unit properties including loss factors.
unit_info = pd.DataFrame({
    'unit': ['A', 'B'],
    'region': ['NSW', 'NSW'],  # MW
    'loss_factor': [0.9, 0.95]
})

# The demand in the region\s being dispatched.
demand = pd.DataFrame({
    'region': ['NSW'],
    'demand': [100.0]  # MW
})

# Create the market model
market = markets.SpotMarket(unit_info=unit_info,
                            market_regions=['NSW'])
market.set_unit_volume_bids(volume_bids)
market.set_unit_price_bids(price_bids)
market.set_unit_bid_capacity_constraints(
    unit_limits.loc[:, ['unit', 'capacity']])
```

(continues on next page)

(continued from previous page)

```
market.set_unit_ramp_up_constraints(
    unit_limits.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
market.set_unit_ramp_down_constraints(
    unit_limits.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
market.set_demand_constraints(demand)

# Calculate dispatch and pricing
market.dispatch()

# Return the total dispatch of each unit in MW.
print(market.get_unit_dispatch())
#   unit service  dispatch
# 0    A  energy      40.0
# 1    B  energy      60.0

# Understanding the dispatch results: In this example unit loss factors are
# provided, that means the cost of a bid in the dispatch optimisation is
# the bid price divided by the unit loss factor. However, loss factors do
# not effect the amount of generation a unit can supply, this is because the
# regional demand already factors in intra regional losses. The cheapest bid is
# from unit A with 20 MW at 44.44 $/MW (after loss factor), this will be
# fully dispatched. The next cheapest bid is from unit B with 50 MW at
# 52.63 $/MW, again fully dispatch. The next cheapest is unit B with 30 MW at
# 57.89 $/MW, however, unit B starts the interval at a dispatch level of 0.0 MW
# and can ramp at speed of 720 MW/hr, the default dispatch interval of Nempy
# is 5 min, so unit B can at most produce 60 MW by the end of the
# dispatch interval, this means only 10 MW of the second bid from unit B can be
# dispatched. Finally, the last bid that needs to be dispatch for supply to
# equal demand is from unit A with 25 MW at 66.67 $/MW, only 20 MW of this
# bid is needed. Adding together the bids from each unit we can see that
# unit A is dispatch for a total of 40 MW and unit B for a total of 60 MW.

# Return the price of energy in each region.
print(market.get_energy_prices())
#   region  price
# 0    NSW  66.67

# Understanding the pricing result: In this case the marginal bid, the bid
# that would be dispatch if demand increased is the second bid from unit A,
# after adjusting for the loss factor this bid has a price of 66.67 $/MW bid,
# and this bid sets the price.
```

3.3 3. Interconnector with losses

A simple example demonstrating how to implement a two region market with an interconnector. The interconnector is modelled simply, with a fixed percentage of losses. To make the interconnector flow and loss calculation easy to understand a single unit is modelled in the NSW region, NSW demand is set zero, and VIC region demand is set to 90 MW, thus all the power to meet VIC demand must flow across the interconnetcor.

```
import pandas as pd
from nempy import markets

# The only generator is located in NSW.
unit_info = pd.DataFrame({
    'unit': ['A'],
    'region': ['NSW']  # MW
})

# Create a market instance.
market = markets.SpotMarket(unit_info=unit_info, market_regions=['NSW', 'VIC'])

# Volume of each bids.
volume_bids = pd.DataFrame({
    'unit': ['A'],
    '1': [100.0]  # MW
})

market.set_unit_volume_bids(volume_bids)

# Price of each bid.
price_bids = pd.DataFrame({
    'unit': ['A'],
    '1': [50.0]  # $/MW
})

market.set_unit_price_bids(price_bids)

# NSW has no demand but VIC has 90 MW.
demand = pd.DataFrame({
    'region': ['NSW', 'VIC'],
    'demand': [0.0, 90.0]  # MW
})

market.set_demand_constraints(demand)

# There is one interconnector between NSW and VIC. Its nominal direction is towards VIC.
interconnectors = pd.DataFrame({
    'interconnector': ['little_link'],
    'to_region': ['VIC'],
    'from_region': ['NSW'],
    'max': [100.0],
    'min': [-120.0]
})

market.set_interconnectors(interconnectors)
```

(continues on next page)

(continued from previous page)

```


# The interconnector loss function. In this case losses are always 5 % of line flow.
def constant_losses(flow):
    return abs(flow) * 0.05


# The loss function on a per interconnector basis. Also details how the losses should be␣
↪proportioned to the
# connected regions.
loss_functions = pd.DataFrame({
    'interconnector': ['little_link'],
    'from_region_loss_share': [0.5],  # losses are shared equally.
    'loss_function': [constant_losses]
})

# The points to linearly interpolate the loss function between. In this example the loss␣
↪function is linear so only
# three points are needed, but if a non linear loss function was used then more points␣
↪would be better.
interpolation_break_points = pd.DataFrame({
    'interconnector': ['little_link', 'little_link', 'little_link'],
    'loss_segment': [1, 2, 3],
    'break_point': [-120.0, 0.0, 100]
})

market.set_interconnector_losses(loss_functions, interpolation_break_points)

# Calculate dispatch.
market.dispatch()

# Return interconnector flow and losses.
print(market.get_interconnector_flows())
#   interconnector       flow    losses
# 0    little_link  92.307692  4.615385

# Understanding the interconnector flows: Losses are modelled as extra demand
# in the regions on either side of the interconnector, in this case the losses
# are split evenly between the regions. All demand in VIC must be supplied
# across the interconnector, and losses in VIC will add to the interconnector
# flow required, so we can write the equation:
#
# flow = vic_demand + flow * loss_pct_vic
# flow - flow * loss_pct_vic = vic_demand
# flow * (1 - loss_pct_vic) = vic_demand
# flow = vic_demand / (1 - loss_pct_vic)
#
# Since interconnector losses are 5% and half occur in VIC the
# loss_pct_vic = 2.5%. Thus:
#
# flow = 90 / (1 - 0.025) = 92.31
#
```

(continues on next page)

(continued from previous page)

```
# Knowing the interconnector flow we can work out total losses
#
# losses = flow * loss_pct
# losses = 92.31 * 0.05 = 4.62

# Return the total dispatch of each unit in MW.
print(market.get_unit_dispatch())
#   unit service   dispatch
# 0    A  energy  94.615385

# Understanding dispatch results: Unit A must be dispatch to
# meet supply in VIC plus all interconnector losses, therefore
# dispatch is 90.0 + 4.62 = 94.62.

# Return the price of energy in each region.
print(market.get_energy_prices())
#   region      price
# 0    NSW  50.000000
# 1    VIC  52.564103

# Understanding pricing results: The marginal cost of supply in NSW is simply
# the cost of unit A's bid. However, the marginal cost of supply in VIC also
# includes the cost of paying for interconnector losses.
```

3.4 4. Dynamic non-linear interconnector losses

This example demonstrates how to model regional demand dependant interconnector loss functions as decribed in the AEMO `Marginal Loss Factors documentation section 3 to 5`. To make the interconnector flow and loss calculation easy to understand a single unit is modelled in the NSW region, NSW demand is set zero, and VIC region demand is set to 800 MW, thus all the power to meet VIC demand must flow across the interconnetcor.

```
import pandas as pd
from nempy import markets
from nempy.historical_inputs import interconnectors as interconnector_inputs


# The only generator is located in NSW.
unit_info = pd.DataFrame({
    'unit': ['A'],
    'region': ['NSW']  # MW
})

# Create a market instance.
market = markets.SpotMarket(unit_info=unit_info,
                            market_regions=['NSW', 'VIC'])

# Volume of each bids.
volume_bids = pd.DataFrame({
    'unit': ['A'],
    '1': [1000.0]  # MW
```

(continues on next page)

(continued from previous page)

```python
})

market.set_unit_volume_bids(volume_bids)

# Price of each bid.
price_bids = pd.DataFrame({
    'unit': ['A'],
    '1': [50.0]  # $/MW
})

market.set_unit_price_bids(price_bids)

# NSW has no demand but VIC has 800 MW.
demand = pd.DataFrame({
    'region': ['NSW', 'VIC'],
    'demand': [0.0, 800.0],  # MW
    'loss_function_demand': [0.0, 800.0]  # MW
})

market.set_demand_constraints(demand.loc[:, ['region', 'demand']])

# There is one interconnector between NSW and VIC.
# Its nominal direction is towards VIC.
interconnectors = pd.DataFrame({
    'interconnector': ['VIC1-NSW1'],
    'to_region': ['VIC'],
    'from_region': ['NSW'],
    'max': [1000.0],
    'min': [-1200.0]
})

market.set_interconnectors(interconnectors)

# Create a demand dependent loss function.
# Specify the demand dependency
demand_coefficients = pd.DataFrame({
    'interconnector': ['VIC1-NSW1', 'VIC1-NSW1'],
    'region': ['NSW1', 'VIC1'],
    'demand_coefficient': [0.000021734, -0.000031523]})

# Specify the loss function constant and flow coefficient.
interconnector_coefficients = pd.DataFrame({
    'interconnector': ['VIC1-NSW1'],
    'loss_constant': [1.0657],
    'flow_coefficient': [0.00017027],
    'from_region_loss_share': [0.5]})

# Create loss functions on per interconnector basis.
loss_functions = interconnector_inputs.create_loss_functions(
    interconnector_coefficients, demand_coefficients,
    demand.loc[:, ['region', 'loss_function_demand']])

```

(continues on next page)

(continued from previous page)

```
# The points to linearly interpolate the loss function between.
interpolation_break_points = pd.DataFrame({
    'interconnector': 'VIC1-NSW1',
    'loss_segment': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
    'break_point': [-1200.0, -1000.0, -800.0, -600.0, -400.0, -200.0,
                    0.0, 200.0, 400.0, 600.0, 800.0, 1000]
})

market.set_interconnector_losses(loss_functions,
                                 interpolation_break_points)

# Calculate dispatch.
market.dispatch()

# Return interconnector flow and losses.
print(market.get_interconnector_flows())
#   interconnector        flow      losses
# 0      VIC1-NSW1  860.102737  120.205473

# Understanding the interconnector flows: In this case it is not simple to
# analytically derive and explain the interconnector flow result. The loss
# model is constructed within the underlying mixed integer linear problem
# as set of constraints and the interconnector flow and losses are
# determined as part of the problem solution. However, the loss model can
# be explained at a high level, and the results shown to be consistent. The
# first step in the interconnector model is to drive the loss function as a
# function  of regional demand, which is a pre-market model creation step, the
# mathematics is explained in
# docs/pdfs/Marginal Loss Factors for the 2020-21 Financial year.pdf. The loss
# function is then evaluated at the given break points and linearly interpolated
# between those points in the market model. So for our model the losses are
# interpolated between 800 MW and 1000 MW. We can show the losses are consistent
# with this approach:
#
# Losses at a flow of 800 MW
print(loss_functions['loss_function'].iloc[0](800))
# 107.0464
# Losses at a flow of 1000 MW
print(loss_functions['loss_function'].iloc[0](1000))
# 150.835
# Then interpolating by taking the weighted sum of the two losses based on the
# relative difference between the actual flow and the interpolation break points:
# Weighting of 800 MW break point = 1 - ((860.102737 - 800.0)/(1000 - 800))
# Weighting of 800 MW break point = 0.7
# Weighting of 1000 MW break point = 1 - ((1000 - 860.102737)/(1000 - 800))
# Weighting of 1000 MW break point = 0.3
# Weighed sum of losses = 107.0464 * 0.7 + 150.835 * 0.3 = 120.18298
#
# We can also see that the flow and loss results are consistent with the supply
# equals demand constraint, all demand in the VIC region is supplied by the
# interconnector, so the interconnector flow minus the VIC region interconnector
# losses should equal the VIC region demand. Note that the VIC region loss
```

(continues on next page)

(continued from previous page)

```python
# share is 50%:
# VIC region demand = interconnector flow - losses * VIC region loss share
# 800 = 860.102737 - 120.205473 * 0.5
# 800 = 800

# Return the total dispatch of each unit in MW.
print(market.get_unit_dispatch())
#   unit service    dispatch
# 0    A  energy  920.205473

# Understanding the dispatch results: Unit A is the only generator and it must
# be dispatch to meet demand plus losses:
# dispatch = VIC region demand + NSW region demand + losses
# dispatch = 920.205473

# Return the price of energy in each region.
print(market.get_energy_prices())
#   region      price
# 0    NSW  50.000000
# 1    VIC  62.292869

# Understanding the pricing results: Pricing in the NSW region is simply the
# marginal cost of supply from unit A. The marginal cost of supply in the
# VIC region is the cost of unit A to meet both marginal demand and the
# marginal losses on the interconnector.
```

3.5 5. Simple FCAS markets

This example implements a market for energy, regulation raise and contingency 6 sec raise, with co-optimisation constraints as described in section 6.2 and 6.3 of `FCAS Model in NEMDE`.

```python
import pandas as pd
from nempy import markets


# Set options so you see all DataFrame columns in print outs.
pd.options.display.width = 0

# Volume of each bid.
volume_bids = pd.DataFrame({
    'unit': ['A', 'A', 'B', 'B', 'B'],
    'service': ['energy', 'raise_6s', 'energy',
                'raise_6s', 'raise_reg'],
    '1': [100.0, 10.0, 110.0, 15.0, 15.0],  # MW
})

print(volume_bids)
#   unit    service      1
# 0    A      energy  100.0
# 1    A    raise_6s   10.0
```

(continues on next page)

(continued from previous page)

```
# 2    B      energy  110.0
# 3    B    raise_6s   15.0
# 4    B  raise_reg    15.0

# Price of each bid.
price_bids = pd.DataFrame({
    'unit': ['A', 'A', 'B', 'B', 'B'],
    'service': ['energy', 'raise_6s', 'energy',
                'raise_6s', 'raise_reg'],
    '1': [50.0, 35.0, 60.0, 20.0, 30.0],  # $/MW
})

print(price_bids)
#   unit    service     1
# 0    A      energy  50.0
# 1    A    raise_6s  35.0
# 2    B      energy  60.0
# 3    B    raise_6s  20.0
# 4    B  raise_reg  30.0

# Participant defined operational constraints on FCAS enablement.
fcas_trapeziums = pd.DataFrame({
    'unit': ['B', 'B', 'A'],
    'service': ['raise_reg', 'raise_6s', 'raise_6s'],
    'max_availability': [15.0, 15.0, 10.0],
    'enablement_min': [50.0, 50.0, 70.0],
    'low_break_point': [65.0, 65.0, 80.0],
    'high_break_point': [95.0, 95.0, 100.0],
    'enablement_max': [110.0, 110.0, 110.0]
})

print(fcas_trapeziums)
#   unit    service  max_availability  enablement_min  low_break_point  high_break_point  enablement_max
# 0    B  raise_reg              15.0            50.0             65.0              95.0           110.0
# 1    B   raise_6s              15.0            50.0             65.0              95.0           110.0
# 2    A   raise_6s              10.0            70.0             80.0             100.0           110.0

# Unit locations.
unit_info = pd.DataFrame({
    'unit': ['A', 'B'],
    'region': ['NSW', 'NSW']
})

print(unit_info)
#   unit region
# 0    A    NSW
# 1    B    NSW

```

(continues on next page)

(continued from previous page)

```python
# The demand in the region\s being dispatched.
demand = pd.DataFrame({
    'region': ['NSW'],
    'demand': [195.0]  # MW
})

print(demand)
#   region  demand
# 0    NSW   195.0

# FCAS requirement in the region\s being dispatched.
fcas_requirements = pd.DataFrame({
    'set': ['nsw_regulation_requirement', 'nsw_raise_6s_requirement'],
    'region': ['NSW', 'NSW'],
    'service': ['raise_reg', 'raise_6s'],
    'volume': [10.0, 10.0]  # MW
})

print(fcas_requirements)
#                           set region    service  volume
# 0  nsw_regulation_requirement    NSW  raise_reg    10.0
# 1    nsw_raise_6s_requirement    NSW   raise_6s    10.0

# Create the market model with unit service bids.
market = markets.SpotMarket(unit_info=unit_info,
                            market_regions=['NSW'])
market.set_unit_volume_bids(volume_bids)
market.set_unit_price_bids(price_bids)

# Create constraints that enforce the top of the FCAS trapezium.
fcas_availability = fcas_trapeziums.loc[:, ['unit', 'service', 'max_availability']]
market.set_fcas_max_availability(fcas_availability)

# Create constraints that enforce the lower and upper slope of the FCAS regulation
# service trapeziums.
regulation_trapeziums = fcas_trapeziums[fcas_trapeziums['service'] == 'raise_reg']
market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)

# Create constraints that enforce the lower and upper slope of the FCAS contingency
# trapezium. These constrains also scale slopes of the trapezium to ensure the
# co-dispatch of contingency and regulation services is technically feasible.
contingency_trapeziums = fcas_trapeziums[fcas_trapeziums['service'] == 'raise_6s']
market.set_joint_capacity_constraints(contingency_trapeziums)

# Set the demand for energy.
market.set_demand_constraints(demand)

# Set the required volume of FCAS services.
market.set_fcas_requirements_constraints(fcas_requirements)

# Calculate dispatch and pricing
market.dispatch()
```

(continues on next page)

(continued from previous page)

```python

# Return the total dispatch of each unit in MW.
print(market.get_unit_dispatch())
#   unit    service  dispatch
# 0    A     energy     100.0
# 1    A   raise_6s       5.0
# 2    B     energy      95.0
# 3    B   raise_6s       5.0
# 4    B  raise_reg      10.0

# Understanding the dispatch results: Starting with the raise regulation
# service we can see that only unit B has bid to provide this service so
# 10 MW of it's raise regulation bid must be dispatch. For the raise 6 s
# service while unit B is cheaper it's provision of 10 MW of raise
# regulation means it can only provide 5 MW of raise 6 s, so 5 MW must be
# provided by unit A. For the energy service unit A is cheaper so all
# 100 MW of it's energy bid are dispatched, leaving the remaining 95 MW to
# provided by unit B. Also, note that these energy and FCAS dispatch levels are
# permitted by the FCAS trapezium constraints. Further explanation of these
# constraints are provided here: docs/pdfs/FCAS Model in NEMDE.pdf.

# Return the price of energy.
print(market.get_energy_prices())
#   region  price
# 0    NSW   75.0

#  Understanding energy price results:
#  A marginal unit of energy would have to come from unit B, as unit A is fully
#  dispatch, this would cost 60 $/MW/h. However, to turn unit B up, you would
#  need it to dispatch less raise_6s, this would cost - 20 $/MW/h, and the
#  extra FCAS would have to come from unit A, this would cost 35 $/MW/h.
#  Therefore the marginal cost of energy is 60 - 20 + 35 = 75 $/MW/h

# Return the price of regulation FCAS.
print(market.get_fcas_prices())
#   region    service  price
# 0    NSW   raise_6s   35.0
# 1    NSW  raise_reg   45.0

# Understanding FCAS price results:
# A marginal unit of raise_reg would have to come from unit B as it is the only
# provider, this would cost 30 $/MW/h. It would also require unit B to provide
# less raise_6s, this would cost -20 $/MW/h, extra raise_6s would then be
# required from unit A costing 35 $/MW/h. This gives a total marginal cost of
# 30 - 20 + 35 = 45 $/MW/h.
#
# A marginal unit of raise_6s would be provided by unit A at a cost of 35$/MW/h/.
```

3.6 6. Simple recreation of historical dispatch

Demonstrates using Nempy to recreate historical dispatch intervals by implementing a simple energy market with unit bids, unit maximum capacity constraints and interconnector models, all sourced from historical data published by AEMO.

Results from example: for the QLD region a reasonable fit between modelled prices and historical prices is obtained.

> **Warning:** Warning this script downloads approximately 8.5 GB of data from AEMO. The download_inputs flag can be set to false to stop the script re-downloading data for subsequent runs.

Note: This example also requires plotly >= 5.3.1, < 6.0.0 and kaleido == 0.2.1. Run pip install plotly==5.3.1 and pip install kaleido==0.2.1

```python
# Notice:
# - This script downloads large volumes of historical market data from AEMO's nemweb
#   portal. The boolean on line 20 can be changed to prevent this happening repeatedly
#   once the data has been downloaded.
# - This example also requires plotly >= 5.3.1, < 6.0.0 and kaleido == 0.2.1
#   pip install plotly==5.3.1 and pip install kaleido==0.2.1

import sqlite3
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
import plotly.graph_objects as go
from nempy import markets
from nempy.historical_inputs import loaders, mms_db, \
    xml_cache, units, demand, interconnectors

con = sqlite3.connect('historical_mms.db')
mms_db_manager = mms_db.DBManager(connection=con)

xml_cache_manager = xml_cache.XMLCacheManager('nemde_cache')

# The second time this example is run on a machine this flag can
# be set to false to save downloading the data again.
download_inputs = True

if download_inputs:
    # This requires approximately 5 GB of storage.
    mms_db_manager.populate(start_year=2019, start_month=1,
                            end_year=2019, end_month=1)

    # This requires approximately 3.5 GB of storage.
    xml_cache_manager.populate_by_day(start_year=2019, start_month=1, start_day=1,
                                      end_year=2019, end_month=1, end_day=1)

raw_inputs_loader = loaders.RawInputsLoader(
    nemde_xml_cache_manager=xml_cache_manager,
    market_management_system_database=mms_db_manager)

# A list of intervals we want to recreate historical dispatch for.
dispatch_intervals = ['2019/01/01 12:00:00',
                      '2019/01/01 12:05:00',
                      '2019/01/01 12:10:00',
                      '2019/01/01 12:15:00',
                      '2019/01/01 12:20:00',
                      '2019/01/01 12:25:00',
                      '2019/01/01 12:30:00']

# List for saving outputs to.
outputs = []

# Create and dispatch the spot market for each dispatch interval.
for interval in dispatch_intervals:
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)
    interconnector_inputs = \
        interconnectors.InterconnectorData(raw_inputs_loader)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)
```

(continues on next page)

(continued from previous page)

```python

    volume_bids, price_bids = unit_inputs.get_processed_bids()
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)

    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
    market.set_unconstrained_intermitent_generation_forecast_constraint(
        unit_uigf_limit)

    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)

    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)
    market.dispatch()

    # Save prices from this interval
    prices = market.get_energy_prices()
    prices['time'] = interval

    # Getting historical prices for comparison. Note, ROP price, which is
    # the regional reference node price before the application of any
    # price scaling by AEMO, is used for comparison.
    historical_prices = mms_db_manager.DISPATCHPRICE.get_data(interval)

    prices = pd.merge(prices, historical_prices,
                      left_on=['time', 'region'],
                      right_on=['SETTLEMENTDATE', 'REGIONID'])

    outputs.append(
        prices.loc[:, ['time', 'region', 'price', 'ROP']])

con.close()

outputs = pd.concat(outputs)

# Plot results for QLD market region.
qld_prices = outputs[outputs['region'] == 'QLD1']

fig = go.Figure()
fig.add_trace(go.Scatter(x=qld_prices['time'], y=qld_prices['price'], name='Nempy price', mode='markers',
                         marker_size=12, marker_symbol='circle'))
fig.add_trace(go.Scatter(x=qld_prices['time'], y=qld_prices['ROP'], name='Historical
```

(continues on next page)

(continued from previous page)

```
↪price', mode='markers',
                         marker_size=8))
fig.update_xaxes(title="Time")
fig.update_yaxes(title="Price ($/MWh)")
fig.update_layout(yaxis_range=[0.0, 100.0], title="QLD Region Price")
fig.write_image('energy_market_only_qld_prices.png')
fig.show()

print(outputs)
#                     time region      price       ROP
# 0  2019/01/01 12:00:00   NSW1  91.857666  91.87000
# 1  2019/01/01 12:00:00   QLD1  76.180429  76.19066
# 2  2019/01/01 12:00:00    SA1  85.126914  86.89938
# 3  2019/01/01 12:00:00   TAS1  85.948523  89.70523
# 4  2019/01/01 12:00:00   VIC1  83.250703  84.98410
# 0  2019/01/01 12:05:00   NSW1  88.357224  91.87000
# 1  2019/01/01 12:05:00   QLD1  72.255334  64.99000
# 2  2019/01/01 12:05:00    SA1  82.417720  87.46213
# 3  2019/01/01 12:05:00   TAS1  83.451561  90.08096
# 4  2019/01/01 12:05:00   VIC1  80.621103  85.55555
# 0  2019/01/01 12:10:00   NSW1  91.857666  91.87000
# 1  2019/01/01 12:10:00   QLD1  75.665675  64.99000
# 2  2019/01/01 12:10:00    SA1  85.680310  86.86809
# 3  2019/01/01 12:10:00   TAS1  86.715499  89.87995
# 4  2019/01/01 12:10:00   VIC1  83.774337  84.93569
# 0  2019/01/01 12:15:00   NSW1  88.343034  91.87000
# 1  2019/01/01 12:15:00   QLD1  71.746786  64.78003
# 2  2019/01/01 12:15:00    SA1  82.379539  86.84407
# 3  2019/01/01 12:15:00   TAS1  83.451561  89.48585
# 4  2019/01/01 12:15:00   VIC1  80.621103  84.99034
# 0  2019/01/01 12:20:00   NSW1  91.864122  91.87000
# 1  2019/01/01 12:20:00   QLD1  75.052319  64.78003
# 2  2019/01/01 12:20:00    SA1  85.722028  87.49564
# 3  2019/01/01 12:20:00   TAS1  86.576848  90.28958
# 4  2019/01/01 12:20:00   VIC1  83.859306  85.59438
# 0  2019/01/01 12:25:00   NSW1  91.864122  91.87000
# 1  2019/01/01 12:25:00   QLD1  75.696247  64.99000
# 2  2019/01/01 12:25:00    SA1  85.746024  87.51983
# 3  2019/01/01 12:25:00   TAS1  86.613642  90.38750
# 4  2019/01/01 12:25:00   VIC1  83.894945  85.63046
# 0  2019/01/01 12:30:00   NSW1  91.870167  91.87000
# 1  2019/01/01 12:30:00   QLD1  75.188735  64.99000
# 2  2019/01/01 12:30:00    SA1  85.694071  87.46153
# 3  2019/01/01 12:30:00   TAS1  86.560602  90.09919
# 4  2019/01/01 12:30:00   VIC1  83.843570  85.57286
```

3.7 7. Detailed recreation of historical dispatch with Basslink switch run

This example demonstrates using Nempy to recreate historical dispatch intervals by implementing an energy market using all the features of the Nempy market model, with inputs sourced from historical data published by AEMO. This example has been updated to include the use of functionality developed to enable modelling the Basslink switch run, which is new in Nempy version 2.0.0. Previously, Nempy relied on using the generic constraint RHS values reported with the NEMDE solution from what historically was the least cost case of the switch run. However, the new functionality allows the RHS values for each case of the switch run to be calculated by Nempy, and so for each case of switch run to be tested.

> **Warning:** Warning this script downloads approximately 54 GB of data from AEMO. The download_inputs flag can be set to false to stop the script re-downloading data for subsequent runs.

```
1  # Notice:
2  # - This script downloads large volumes of historical market data (~54 GB) from AEMO's nemweb
3  #   portal. You can also reduce the data usage by restricting the time window given to the
4  #   xml_cache_manager and in the get_test_intervals function. The boolean on line 23 can
5  #   also be changed to prevent this happening repeatedly once the data has been downloaded.
6  
7  import sqlite3
8  from datetime import datetime, timedelta
9  import random
10 import pandas as pd
11 from nempy import markets
12 from nempy.historical_inputs import loaders, mms_db, \
13     xml_cache, units, demand, interconnectors, constraints, rhs_calculator
14 from nempy.help_functions.helper_functions import update_rhs_values
15 
16 con = sqlite3.connect('D:/nempy_2021/historical_mms.db')
17 mms_db_manager = mms_db.DBManager(connection=con)
18 
19 xml_cache_manager = xml_cache.XMLCacheManager('D:/nempy_2021/xml_cache')
20 
21 # The second time this example is run on a machine this flag can
22 # be set to false to save downloading the data again.
23 download_inputs = True
24 
25 if download_inputs:
26     # This requires approximately 4 GB of storage.
27     mms_db_manager.populate(start_year=2021, start_month=1,
28                             end_year=2021, end_month=1)
29 
30     # This requires approximately 50 GB of storage.
31     xml_cache_manager.populate_by_day(start_year=2021, start_month=1, start_day=1,
32                                       end_year=2021, end_month=2, end_day=1)
33 
```

(continues on next page)

(continued from previous page)

```
raw_inputs_loader = loaders.RawInputsLoader(
    nemde_xml_cache_manager=xml_cache_manager,
    market_management_system_database=mms_db_manager)


# A list of intervals we want to recreate historical dispatch for.
def get_test_intervals(number=100):
    start_time = datetime(year=2021, month=12, day=1, hour=0, minute=0)
    end_time = datetime(year=2021, month=12, day=31, hour=0, minute=0)
    difference = end_time - start_time
    difference_in_5_min_intervals = difference.days * 12 * 24
    random.seed(1)
    intervals = random.sample(range(1, difference_in_5_min_intervals), number)
    times = [start_time + timedelta(minutes=5 * i) for i in intervals]
    times_formatted = [t.isoformat().replace('T', ' ').replace('-', '/') for t in times]
    return times_formatted


# List for saving outputs to.
outputs = []
c = 0
# Create and dispatch the spot market for each dispatch interval.
for interval in get_test_intervals(number=100):
    c += 1
    print(str(c) + ' ' + str(interval))
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    interconnector_inputs = interconnectors.InterconnectorData(raw_inputs_loader)
    constraint_inputs = constraints.ConstraintData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)
    rhs_calculation_engine = rhs_calculator.RHSCalc(xml_cache_manager)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)

    # Set bids
    volume_bids, price_bids = unit_inputs.get_processed_bids()
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    # Set bid in capacity limits
    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['unit_capacity']
    market.make_constraints_elastic('unit_bid_capacity', violation_cost=cost)

    # Set limits provided by the unconstrained intermittent generation
    # forecasts. Primarily for wind and solar.
    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
    market.set_unconstrained_intermitent_generation_forecast_constraint(
```

(continues on next page)

(continued from previous page)

```
        unit_uigf_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['uigf']
    market.make_constraints_elastic('uigf_capacity', violation_cost=cost)

    # Set unit ramp rates.
    def set_ramp_rates(run_type):
        ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch(run_type=run_
↪type)
        market.set_unit_ramp_up_constraints(
            ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
        market.set_unit_ramp_down_constraints(
            ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
        cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
        market.make_constraints_elastic('ramp_up', violation_cost=cost)
        market.make_constraints_elastic('ramp_down', violation_cost=cost)


    set_ramp_rates(run_type='fast_start_first_run')

    # Set unit FCAS trapezium constraints.
    unit_inputs.add_fcas_trapezium_constraints()
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_max_avail']
    fcas_availability = unit_inputs.get_fcas_max_availability()
    market.set_fcas_max_availability(fcas_availability)
    market.make_constraints_elastic('fcas_max_availability', cost)
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
    regulation_trapeziums = unit_inputs.get_fcas_regulation_trapeziums()
    market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)
    market.make_constraints_elastic('energy_and_regulation_capacity', cost)
    contingency_trapeziums = unit_inputs.get_contingency_services()
    market.set_joint_capacity_constraints(contingency_trapeziums)
    market.make_constraints_elastic('joint_capacity', cost)


    def set_joint_ramping_constraints(run_type):
        cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
        scada_ramp_down_rates = unit_inputs.get_scada_ramp_down_rates_of_lower_reg_units(
            run_type=run_type)
        market.set_joint_ramping_constraints_lower_reg(scada_ramp_down_rates)
        market.make_constraints_elastic('joint_ramping_lower_reg', cost)
        scada_ramp_up_rates = unit_inputs.get_scada_ramp_up_rates_of_raise_reg_units(
            run_type=run_type)
        market.set_joint_ramping_constraints_raise_reg(scada_ramp_up_rates)
        market.make_constraints_elastic('joint_ramping_raise_reg', cost)


    set_joint_ramping_constraints(run_type="fast_start_first_run")

    # Set interconnector definitions, limits and loss models.
    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
```

(continues on next page)

(continued from previous page)

```
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)

    # Calculate rhs constraint values that depend on the basslink frequency controller from scratch so there is
    # consistency between the basslink switch runs.
    # Find the constraints that need to be calculated because they depend on the frequency controller status.
    constraints_to_update = (
        rhs_calculation_engine.get_rhs_constraint_equations_that_depend_value('BL_FREQ_ONSTATUS', 'W'))
    initial_bl_freq_onstatus = rhs_calculation_engine.scada_data['W']['BL_FREQ_ONSTATUS'][0]['@Value']
    # Calculate new rhs values for the constraints that need updating.
    new_rhs_values = rhs_calculation_engine.compute_constraint_rhs(constraints_to_update)

    # Add generic constraints and FCAS market constraints.
    fcas_requirements = constraint_inputs.get_fcas_requirements()
    fcas_requirements = update_rhs_values(fcas_requirements, new_rhs_values)
    market.set_fcas_requirements_constraints(fcas_requirements)
    violation_costs = constraint_inputs.get_violation_costs()
    market.make_constraints_elastic('fcas', violation_cost=violation_costs)
    generic_rhs = constraint_inputs.get_rhs_and_type_excluding_regional_fcas_constraints()
    generic_rhs = update_rhs_values(generic_rhs, new_rhs_values)
    market.set_generic_constraints(generic_rhs)
    market.make_constraints_elastic('generic', violation_cost=violation_costs)

    unit_generic_lhs = constraint_inputs.get_unit_lhs()
    market.link_units_to_generic_constraints(unit_generic_lhs)
    interconnector_generic_lhs = constraint_inputs.get_interconnector_lhs()
    market.link_interconnectors_to_generic_constraints(
        interconnector_generic_lhs)

    # Set the operational demand to be met by dispatch.
    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)

    # Set tiebreak constraint to equalise dispatch of equally priced bids.
    cost = constraint_inputs.get_constraint_violation_prices()['tiebreak']
    market.set_tie_break_constraints(cost)

    # Get unit dispatch without fast start constraints and use it to
    # make fast start unit commitment decisions.
    market.dispatch()
    dispatch = market.get_unit_dispatch()
    fast_start_profiles = unit_inputs.get_fast_start_profiles_for_dispatch(dispatch)
    set_ramp_rates(run_type='fast_start_second_run')
    set_joint_ramping_constraints(run_type='fast_start_second_run')
    market.set_fast_start_constraints(fast_start_profiles)
```

(continues on next page)

(continued from previous page)

```
    if 'fast_start' in market.get_constraint_set_names.keys():
        cost = constraint_inputs.get_constraint_violation_prices()['fast_start']
        market.make_constraints_elastic('fast_start', violation_cost=cost)

    # First run of Basslink switch runs
    market.dispatch()  # First dispatch without allowing over constrained dispatch re-run to get objective function.
    objective_value_run_one = market.objective_value
    if constraint_inputs.is_over_constrained_dispatch_rerun():
        market.dispatch(allow_over_constrained_dispatch_re_run=True,
                        energy_market_floor_price=-1000.0,
                        energy_market_ceiling_price=15000.0,
                        fcas_market_ceiling_price=1000.0)
    prices_run_one = market.get_energy_prices()  # If this is the lowest cost run these will be the market prices.

    # Re-run dispatch with Basslink Frequency controller off.
    # Set frequency controller to off in rhs calculations
    rhs_calculation_engine.update_spd_id_value('BL_FREQ_ONSTATUS', 'W', '0')
    new_bl_freq_onstatus = rhs_calculation_engine.scada_data['W']['BL_FREQ_ONSTATUS'][0]['@Value']
    # Find the constraints that need to be updated because they depend on the frequency controller status.
    constraints_to_update = (
        rhs_calculation_engine.get_rhs_constraint_equations_that_depend_value('BL_FREQ_ONSTATUS', 'W'))
    # Calculate new rhs values for the constraints that need updating.
    new_rhs_values = rhs_calculation_engine.compute_constraint_rhs(constraints_to_update)
    # Update the constraints in the market.
    fcas_requirements = update_rhs_values(fcas_requirements, new_rhs_values)
    violation_costs = constraint_inputs.get_violation_costs()
    market.set_fcas_requirements_constraints(fcas_requirements)
    market.make_constraints_elastic('fcas', violation_cost=violation_costs)
    generic_rhs = update_rhs_values(generic_rhs, new_rhs_values)
    market.set_generic_constraints(generic_rhs)
    market.make_constraints_elastic('generic', violation_cost=violation_costs)

    # Reset ramp rate constraints for first run of second Basslink switchrun
    set_ramp_rates(run_type='fast_start_first_run')
    set_joint_ramping_constraints(run_type='fast_start_first_run')

    # Get unit dispatch without fast start constraints and use it to
    # make fast start unit commitment decisions.
    market.remove_fast_start_constraints()
    market.dispatch()
    dispatch = market.get_unit_dispatch()
    fast_start_profiles = unit_inputs.get_fast_start_profiles_for_dispatch(dispatch)
    set_ramp_rates(run_type='fast_start_second_run')
    set_joint_ramping_constraints(run_type='fast_start_second_run')
    market.set_fast_start_constraints(fast_start_profiles)
    if 'fast_start' in market.get_constraint_set_names():
        cost = constraint_inputs.get_constraint_violation_prices()['fast_start']
```

(continues on next page)

(continued from previous page)

```
        market.make_constraints_elastic('fast_start', violation_cost=cost)

    market.dispatch()  # First dispatch without allowing over constrained dispatch re-
↪run to get objective function.
    objective_value_run_two = market.objective_value
    if constraint_inputs.is_over_constrained_dispatch_rerun():
        market.dispatch(allow_over_constrained_dispatch_re_run=True,
                        energy_market_floor_price=-1000.0,
                        energy_market_ceiling_price=15000.0,
                        fcas_market_ceiling_price=1000.0)
    prices_run_two = market.get_energy_prices()  # If this is the lowest cost run these␣
↪will be the market prices.

    prices_run_one['time'] = interval
    prices_run_two['time'] = interval

    # Getting historical prices for comparison. Note, ROP price, which is
    # the regional reference node price before the application of any
    # price scaling by AEMO, is used for comparison.
    historical_prices = mms_db_manager.DISPATCHPRICE.get_data(interval)

    # The prices from the run with the lowest objective function value are used.
    if objective_value_run_one < objective_value_run_two:
        prices = prices_run_one
    else:
        prices = prices_run_two

    prices['time'] = interval
    prices = pd.merge(prices, historical_prices,
                      left_on=['time', 'region'],
                      right_on=['SETTLEMENTDATE', 'REGIONID'])

    outputs.append(prices)

con.close()

outputs = pd.concat(outputs)

outputs['error'] = outputs['price'] - outputs['ROP']

print('\n Summary of error in energy price volume weighted average price. \n'
      'Comparison is against ROP, the price prior to \n'
      'any post dispatch adjustments, scaling, capping etc.')
print('Mean price error: {}'.format(outputs['error'].mean()))
print('Median price error: {}'.format(outputs['error'].quantile(0.5)))
print('5% percentile price error: {}'.format(outputs['error'].quantile(0.05)))
print('95% percentile price error: {}'.format(outputs['error'].quantile(0.95)))

#  Summary of error in energy price volume weighted average price.
# Comparison is against ROP, the price prior to
# any post dispatch adjustments, scaling, capping etc.
# Mean price error: -0.3284696359015098
```

(continues on next page)

(continued from previous page)

```
# Median price error: 0.0
# 5% percentile price error: -0.5389930178124978
# 95% percentile price error: 0.13746097842649457
```

3.8 7. Recreation of historical dispatch without Basslink switchrun

This example demonstrates using Nempy to recreate historical dispatch intervals by implementing an energy market using all the features of the Nempy market model, except the Basslink switch run, with inputs sourced from historical data published by AEMO. The main reason not to include Basslink switch run is to speed up runtime. Note each interval is dispatched as a standalone simulation and the results from one dispatch interval are not carried over to be the initial conditions of the next interval, rather the historical initial conditions are always used.

> **Warning:** Warning this script downloads approximately 54 GB of data from AEMO. The download_inputs flag can be set to false to stop the script re-downloading data for subsequent runs.

```
# Notice:
# - This script downloads large volumes of historical market data (~54 GB) from AEMO's␣
↪nemweb
#   portal. You can also reduce the data usage by restricting the time window given to␣
↪the
#   xml_cache_manager and in the get_test_intervals function. The boolean on line 22 can
#   also be changed to prevent this happening repeatedly once the data has been␣
↪downloaded.

import sqlite3
from datetime import datetime, timedelta
import random
import pandas as pd
from nempy import markets
from nempy.historical_inputs import loaders, mms_db, \
    xml_cache, units, demand, interconnectors, constraints

con = sqlite3.connect('D:/nempy_2021/historical_mms.db')
mms_db_manager = mms_db.DBManager(connection=con)

xml_cache_manager = xml_cache.XMLCacheManager('D:/nempy_2021/xml_cache')

# The second time this example is run on a machine this flag can
# be set to false to save downloading the data again.
download_inputs = True

if download_inputs:
    # This requires approximately 4 GB of storage.
    mms_db_manager.populate(start_year=2021, start_month=1,
                            end_year=2021, end_month=1)

    # This requires approximately 50 GB of storage.
    xml_cache_manager.populate_by_day(start_year=2021, start_month=1, start_day=1,
```

(continues on next page)

(continued from previous page)

```
                                              end_year=2021, end_month=2, end_day=1)

raw_inputs_loader = loaders.RawInputsLoader(
    nemde_xml_cache_manager=xml_cache_manager,
    market_management_system_database=mms_db_manager)


# A list of intervals we want to recreate historical dispatch for.
def get_test_intervals(number=100):
    start_time = datetime(year=2021, month=12, day=1, hour=0, minute=0)
    end_time = datetime(year=2021, month=12, day=31, hour=0, minute=0)
    difference = end_time - start_time
    difference_in_5_min_intervals = difference.days * 12 * 24
    random.seed(1)
    intervals = random.sample(range(1, difference_in_5_min_intervals), number)
    times = [start_time + timedelta(minutes=5 * i) for i in intervals]
    times_formatted = [t.isoformat().replace('T', ' ').replace('-', '/') for t in times]
    return times_formatted


# List for saving outputs to.
outputs = []
c = 0
# Create and dispatch the spot market for each dispatch interval.
for interval in get_test_intervals(number=100):
    c += 1
    print(str(c) + ' ' + str(interval))
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    interconnector_inputs = interconnectors.InterconnectorData(raw_inputs_loader)
    constraint_inputs = constraints.ConstraintData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)

    # Set bids
    volume_bids, price_bids = unit_inputs.get_processed_bids()
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    # Set bid in capacity limits
    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['unit_capacity']
    market.make_constraints_elastic('unit_bid_capacity', violation_cost=cost)

    # Set limits provided by the unconstrained intermittent generation
    # forecasts. Primarily for wind and solar.
    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
```

(continues on next page)

(continued from previous page)

```
    market.set_unconstrained_intermitent_generation_forecast_constraint(
        unit_uigf_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['uigf']
    market.make_constraints_elastic('uigf_capacity', violation_cost=cost)

    # Set unit ramp rates.
    ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch(run_type="fast_start_first_run")
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.make_constraints_elastic('ramp_up', violation_cost=cost)
    market.make_constraints_elastic('ramp_down', violation_cost=cost)

    # Set unit FCAS trapezium constraints.
    unit_inputs.add_fcas_trapezium_constraints()
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_max_avail']
    fcas_availability = unit_inputs.get_fcas_max_availability()
    market.set_fcas_max_availability(fcas_availability)
    market.make_constraints_elastic('fcas_max_availability', cost)
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
    regulation_trapeziums = unit_inputs.get_fcas_regulation_trapeziums()
    market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)
    market.make_constraints_elastic('energy_and_regulation_capacity', cost)
    scada_ramp_down_rates = unit_inputs.get_scada_ramp_down_rates_of_lower_reg_units(run_type="fast_start_first_run")
    market.set_joint_ramping_constraints_lower_reg(scada_ramp_down_rates)
    market.make_constraints_elastic('joint_ramping_lower_reg', cost)
    scada_ramp_up_rates = unit_inputs.get_scada_ramp_up_rates_of_raise_reg_units(run_type="fast_start_first_run")
    market.set_joint_ramping_constraints_raise_reg(scada_ramp_up_rates)
    market.make_constraints_elastic('joint_ramping_raise_reg', cost)
    contingency_trapeziums = unit_inputs.get_contingency_services()
    market.set_joint_capacity_constraints(contingency_trapeziums)
    market.make_constraints_elastic('joint_capacity', cost)

    # Set interconnector definitions, limits and loss models.
    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)

    # Add generic constraints and FCAS market constraints.
    fcas_requirements = constraint_inputs.get_fcas_requirements()
    market.set_fcas_requirements_constraints(fcas_requirements)
    violation_costs = constraint_inputs.get_violation_costs()
    market.make_constraints_elastic('fcas', violation_cost=violation_costs)
```

(continues on next page)

(continued from previous page)

```
    generic_rhs = constraint_inputs.get_rhs_and_type_excluding_regional_fcas_
↪constraints()
    market.set_generic_constraints(generic_rhs)
    market.make_constraints_elastic('generic', violation_cost=violation_costs)
    unit_generic_lhs = constraint_inputs.get_unit_lhs()
    market.link_units_to_generic_constraints(unit_generic_lhs)
    interconnector_generic_lhs = constraint_inputs.get_interconnector_lhs()
    market.link_interconnectors_to_generic_constraints(
        interconnector_generic_lhs)

    # Set the operational demand to be met by dispatch.
    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)

    # Set tiebreak constraint to equalise dispatch of equally priced bids.
    cost = constraint_inputs.get_constraint_violation_prices()['tiebreak']
    market.set_tie_break_constraints(cost)

    # Get unit dispatch without fast start constraints and use it to
    # make fast start unit commitment decisions.
    market.dispatch()
    dispatch = market.get_unit_dispatch()

    fast_start_profiles = unit_inputs.get_fast_start_profiles_for_dispatch(dispatch)
    market.set_fast_start_constraints(fast_start_profiles)

    ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch(run_type="fast_
↪start_second_run")
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.make_constraints_elastic('ramp_up', violation_cost=cost)
    market.make_constraints_elastic('ramp_down', violation_cost=cost)

    cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
    scada_ramp_down_rates = unit_inputs.get_scada_ramp_down_rates_of_lower_reg_units(run_
↪type="fast_start_second_run")
    market.set_joint_ramping_constraints_lower_reg(scada_ramp_down_rates)
    market.make_constraints_elastic('joint_ramping_lower_reg', cost)
    scada_ramp_up_rates = unit_inputs.get_scada_ramp_up_rates_of_raise_reg_units(run_
↪type="fast_start_second_run")
    market.set_joint_ramping_constraints_raise_reg(scada_ramp_up_rates)
    market.make_constraints_elastic('joint_ramping_raise_reg', cost)

    if 'fast_start' in market.get_constraint_set_names.keys():
        cost = constraint_inputs.get_constraint_violation_prices()['fast_start']
        market.make_constraints_elastic('fast_start', violation_cost=cost)

    # If AEMO historically used the over constrained dispatch rerun
    # process then allow it to be used in dispatch. This is needed
```

(continues on next page)

(continued from previous page)

```
    # because sometimes the conditions for over constrained dispatch
    # are present but the rerun process isn't used.
    if constraint_inputs.is_over_constrained_dispatch_rerun():
        market.dispatch(allow_over_constrained_dispatch_re_run=True,
                        energy_market_floor_price=-1000.0,
                        energy_market_ceiling_price=15000.0,
                        fcas_market_ceiling_price=1000.0)
    else:
        # The market price ceiling and floor are not needed here
        # because they are only used for the over constrained
        # dispatch rerun process.
        market.dispatch(allow_over_constrained_dispatch_re_run=False)

    # Save prices from this interval
    prices = market.get_energy_prices()
    prices['time'] = interval

    # Getting historical prices for comparison. Note, ROP price, which is
    # the regional reference node price before the application of any
    # price scaling by AEMO, is used for comparison.
    historical_prices = mms_db_manager.DISPATCHPRICE.get_data(interval)

    prices = pd.merge(prices, historical_prices,
                      left_on=['time', 'region'],
                      right_on=['SETTLEMENTDATE', 'REGIONID'])

    outputs.append(
        prices.loc[:, ['time', 'region', 'price', 'ROP']])

con.close()

outputs = pd.concat(outputs)

outputs['error'] = outputs['price'] - outputs['ROP']

print('\n Summary of error in energy price volume weighted average price. \n'
      'Comparison is against ROP, the price prior to \n'
      'any post dispatch adjustments, scaling, capping etc.')
print('Mean price error: {}'.format(outputs['error'].mean()))
print('Median price error: {}'.format(outputs['error'].quantile(0.5)))
print('5% percentile price error: {}'.format(outputs['error'].quantile(0.05)))
print('95% percentile price error: {}'.format(outputs['error'].quantile(0.95)))

#  Summary of error in energy price volume weighted average price.
# Comparison is against ROP, the price prior to
# any post dispatch adjustments, scaling, capping etc.
# Mean price error: -0.3282044852032724
# Median price error: 0.0
# 5% percentile price error: -0.5389930178124978
# 95% percentile price error: 0.13746097842649457
```

3.9 8. Time sequential recreation of historical dispatch

This example demonstrates using Nempy to recreate historical dispatch in a dynamic or time sequential manner, this means the outputs of one interval become the initial conditions for the next dispatch interval. Note, currently there is not the infrastructure in place to include features such as generic constraints in the time sequential model as the rhs values of many constraints would need to be re-calculated based on the dynamic system state. Similarly, using historical bids in this example is some what problematic as participants also dynamically change their bids based on market conditions. However, for the sake of demonstrating how Nempy can be used to create time sequential models, historical bids are used in this example.

> **Warning:** Warning this script downloads approximately 8.5 GB of data from AEMO. The download_inputs flag can be set to false to stop the script re-downloading data for subsequent runs.

```python
# Notice:
# - This script downloads large volumes of historical market data from AEMO's nemweb
#   portal. The boolean on line 20 can be changed to prevent this happening repeatedly
#   once the data has been downloaded.
# - This example also requires plotly >= 5.3.1, < 6.0.0 and kaleido == 0.2.1
#   pip install plotly==5.3.1 and pip install kaleido==0.2.1

import sqlite3
import pandas as pd
from nempy import markets, time_sequential
from nempy.historical_inputs import loaders, mms_db, \
    xml_cache, units, demand, interconnectors, constraints

con = sqlite3.connect('market_management_system.db')
mms_db_manager = mms_db.DBManager(connection=con)

xml_cache_manager = xml_cache.XMLCacheManager('cache_directory')

# The second time this example is run on a machine this flag can
# be set to false to save downloading the data again.
download_inputs = True

if download_inputs:
    # This requires approximately 5 GB of storage.
    mms_db_manager.populate(start_year=2019, start_month=1,
                            end_year=2019, end_month=1)

    # This requires approximately 3.5 GB of storage.
    xml_cache_manager.populate_by_day(start_year=2019, start_month=1, start_day=1,
                                      end_year=2019, end_month=1, end_day=1)

raw_inputs_loader = loaders.RawInputsLoader(
    nemde_xml_cache_manager=xml_cache_manager,
    market_management_system_database=mms_db_manager)

# A list of intervals we want to recreate historical dispatch for.
dispatch_intervals = ['2019/01/01 12:00:00',
                      '2019/01/01 12:05:00',
```

(continues on next page)

(continued from previous page)

```
                        '2019/01/01 12:10:00',
                        '2019/01/01 12:15:00',
                        '2019/01/01 12:20:00',
                        '2019/01/01 12:25:00',
                        '2019/01/01 12:30:00']

# List for saving outputs to.
outputs = []

unit_dispatch = None
from time import time

t0 = time()
# Create and dispatch the spot market for each dispatch interval.
for interval in dispatch_intervals:
    print(interval)
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)
    interconnector_inputs = \
        interconnectors.InterconnectorData(raw_inputs_loader)
    constraint_inputs = \
        constraints.ConstraintData(raw_inputs_loader)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)

    volume_bids, price_bids = unit_inputs.get_processed_bids()
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    violation_cost = \
        constraint_inputs.get_constraint_violation_prices()['unit_capacity']
    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)
    market.make_constraints_elastic('unit_bid_capacity', violation_cost)

    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
    market.set_unconstrained_intermitent_generation_forecast_constraint(
        unit_uigf_limit)

    ramp_rates = unit_inputs.get_as_bid_ramp_rates()

    # This is the part that makes it time sequential.
    if unit_dispatch is None:
        # For the first dispatch interval we use historical values
        # as initial conditions.
        historical_dispatch = unit_inputs.get_initial_unit_output()
        ramp_rates = time_sequential.create_seed_ramp_rate_parameters(
            historical_dispatch, ramp_rates)
```

(continues on next page)

(continued from previous page)

```
    else:
        # For subsequent dispatch intervals we use the output levels
        # determined by the last dispatch as the new initial conditions
        ramp_rates = time_sequential.construct_ramp_rate_parameters(
            unit_dispatch, ramp_rates)

    violation_cost = \
        constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.make_constraints_elastic('ramp_up', violation_cost)
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    market.make_constraints_elastic('ramp_down', violation_cost)

    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)

    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)
    market.dispatch()

    # Save prices from this interval
    prices = market.get_energy_prices()
    prices['time'] = interval
    outputs.append(prices.loc[:, ['time', 'region', 'price']])

    unit_dispatch = market.get_unit_dispatch()
print("Run time per interval {}".format((time()-t0)/len(dispatch_intervals)))
con.close()
print(pd.concat(outputs))
#                   time region      price
# 0  2019/01/01 12:00:00   NSW1  91.857666
# 1  2019/01/01 12:00:00   QLD1  76.716642
# 2  2019/01/01 12:00:00    SA1  85.126914
# 3  2019/01/01 12:00:00   TAS1  86.173481
# 4  2019/01/01 12:00:00   VIC1  83.250703
# 0  2019/01/01 12:05:00   NSW1  88.357224
# 1  2019/01/01 12:05:00   QLD1  72.255334
# 2  2019/01/01 12:05:00    SA1  82.417720
# 3  2019/01/01 12:05:00   TAS1  83.451561
# 4  2019/01/01 12:05:00   VIC1  80.621103
# 0  2019/01/01 12:10:00   NSW1  91.857666
# 1  2019/01/01 12:10:00   QLD1  75.665675
# 2  2019/01/01 12:10:00    SA1  85.680310
# 3  2019/01/01 12:10:00   TAS1  86.715499
# 4  2019/01/01 12:10:00   VIC1  83.774337
```

(continues on next page)

(continued from previous page)

```
# 0  2019/01/01 12:15:00   NSW1  88.113012
# 1  2019/01/01 12:15:00   QLD1  71.559977
# 2  2019/01/01 12:15:00    SA1  82.165045
# 3  2019/01/01 12:15:00   TAS1  83.451561
# 4  2019/01/01 12:15:00   VIC1  80.411187
# 0  2019/01/01 12:20:00   NSW1  91.864122
# 1  2019/01/01 12:20:00   QLD1  75.052319
# 2  2019/01/01 12:20:00    SA1  85.722028
# 3  2019/01/01 12:20:00   TAS1  86.576848
# 4  2019/01/01 12:20:00   VIC1  83.859306
# 0  2019/01/01 12:25:00   NSW1  91.864122
# 1  2019/01/01 12:25:00   QLD1  75.696247
# 2  2019/01/01 12:25:00    SA1  85.746024
# 3  2019/01/01 12:25:00   TAS1  86.613642
# 4  2019/01/01 12:25:00   VIC1  83.894945
# 0  2019/01/01 12:30:00   NSW1  91.870167
# 1  2019/01/01 12:30:00   QLD1  75.188735
# 2  2019/01/01 12:30:00    SA1  85.694071
# 3  2019/01/01 12:30:00   TAS1  86.560602
# 4  2019/01/01 12:30:00   VIC1  83.843570
```

3.10 10. Nempy performance on older data (Jan 2013, without Basslink switch run)

This example demonstrates using Nempy to recreate historical dispatch intervals by implementing an energy market using all the features of the Nempy market model, with inputs sourced from historical data published by AEMO. A set of 100 random dispatch intervals from January 2015 are dispatched and compared to historical results to see how well Nempy performs for replicating older versions of the NEM's dispatch procedure. Comparison is against ROP, the region price prior to any post dispatch adjustments, scaling, capping etc.

Summary of results:

Mean price error: 0.003
Median price error: 0.000
5% percentile price error: 0.000
95% percentile price error: 0.001

> **Warning:** Warning this script downloads approximately 54 GB of data from AEMO. The download_inputs flag can be set to false to stop the script re-downloading data for subsequent runs.

```
# Notice:
# - This script downloads large volumes of historical market data (~54 GB) from AEMO's␣
↪nemweb
#   portal. You can also reduce the data usage by restricting the time window given to␣
↪the
```

(continues on next page)

(continued from previous page)

```
4  #   xml_cache_manager and in the get_test_intervals function. The boolean on line 23 can
5  #   also be changed to prevent this happening repeatedly once the data has been downloaded.
6
7  import sqlite3
8  from datetime import datetime, timedelta
9  import random
10 import pandas as pd
11 from nempy import markets
12 from nempy.historical_inputs import loaders, mms_db, \
13     xml_cache, units, demand, interconnectors, constraints
14
15
16 con = sqlite3.connect('D:/nempy_2013/historical_mms.db')
17 mms_db_manager = mms_db.DBManager(connection=con)
18
19 xml_cache_manager = xml_cache.XMLCacheManager('D:/nempy_2013/xml_cache')
20
21 # The second time this example is run on a machine this flag can
22 # be set to false to save downloading the data again.
23 download_inputs = True
24
25 if download_inputs:
26     # This requires approximately 4 GB of storage.
27     mms_db_manager.populate(start_year=2013, start_month=1,
28                             end_year=2013, end_month=2)
29
30     # This requires approximately 50 GB of storage.
31     xml_cache_manager.populate_by_day(start_year=2013, start_month=1, start_day=1,
32                                       end_year=2013, end_month=2, end_day=1)
33
34 raw_inputs_loader = loaders.RawInputsLoader(
35     nemde_xml_cache_manager=xml_cache_manager,
36     market_management_system_database=mms_db_manager)
37
38
39 # A list of intervals we want to recreate historical dispatch for.
40 def get_test_intervals(number=100):
41     start_time = datetime(year=2013, month=1, day=1, hour=0, minute=0)
42     end_time = datetime(year=2013, month=1, day=31, hour=0, minute=0)
43     difference = end_time - start_time
44     difference_in_5_min_intervals = difference.days * 12 * 24
45     random.seed(2)
46     intervals = random.sample(range(1, difference_in_5_min_intervals), number)
47     times = [start_time + timedelta(minutes=5 * i) for i in intervals]
48     times_formatted = [t.isoformat().replace('T', ' ').replace('-', '/') for t in times]
49     return times_formatted
50
51
52 # List for saving outputs to.
53 outputs = []
54 c = 0
```

(continues on next page)

(continued from previous page)

```
# Create and dispatch the spot market for each dispatch interval.
for interval in get_test_intervals(number=100):
    c += 1
    print(str(c) + ' ' + str(interval))
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    interconnector_inputs = interconnectors.InterconnectorData(raw_inputs_loader)
    constraint_inputs = constraints.ConstraintData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)

    # Set bids
    volume_bids, price_bids = unit_inputs.get_processed_bids()
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    # Set bid in capacity limits
    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['unit_capacity']
    market.make_constraints_elastic('unit_bid_capacity', violation_cost=cost)

    # Set limits provided by the unconstrained intermittent generation
    # forecasts. Primarily for wind and solar.
    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
    market.set_unconstrained_intermitent_generation_forecast_constraint(
        unit_uigf_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['uigf']
    market.make_constraints_elastic('uigf_capacity', violation_cost=cost)

    # Set unit ramp rates.
    ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch(run_type="fast_
↪start_first_run")
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.make_constraints_elastic('ramp_up', violation_cost=cost)
    market.make_constraints_elastic('ramp_down', violation_cost=cost)

    # Set unit FCAS trapezium constraints.
    unit_inputs.add_fcas_trapezium_constraints()
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_max_avail']
    fcas_availability = unit_inputs.get_fcas_max_availability()
    market.set_fcas_max_availability(fcas_availability)
    market.make_constraints_elastic('fcas_max_availability', cost)
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
```

(continues on next page)

(continued from previous page)

```
    regulation_trapeziums = unit_inputs.get_fcas_regulation_trapeziums()
    market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)
    market.make_constraints_elastic('energy_and_regulation_capacity', cost)
    scada_ramp_down_rates = unit_inputs.get_scada_ramp_down_rates_of_lower_reg_units(run_
↪type="fast_start_first_run")
    market.set_joint_ramping_constraints_lower_reg(scada_ramp_down_rates)
    market.make_constraints_elastic('joint_ramping_lower_reg', cost)
    scada_ramp_up_rates = unit_inputs.get_scada_ramp_up_rates_of_raise_reg_units(run_
↪type="fast_start_first_run")
    market.set_joint_ramping_constraints_raise_reg(scada_ramp_up_rates)
    market.make_constraints_elastic('joint_ramping_raise_reg', cost)
    contingency_trapeziums = unit_inputs.get_contingency_services()
    market.set_joint_capacity_constraints(contingency_trapeziums)
    market.make_constraints_elastic('joint_capacity', cost)

    # Set interconnector definitions, limits and loss models.
    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)

    # Add generic constraints and FCAS market constraints.
    fcas_requirements = constraint_inputs.get_fcas_requirements()
    market.set_fcas_requirements_constraints(fcas_requirements)
    violation_costs = constraint_inputs.get_violation_costs()
    market.make_constraints_elastic('fcas', violation_cost=violation_costs)
    generic_rhs = constraint_inputs.get_rhs_and_type_excluding_regional_fcas_
↪constraints()
    market.set_generic_constraints(generic_rhs)
    market.make_constraints_elastic('generic', violation_cost=violation_costs)
    unit_generic_lhs = constraint_inputs.get_unit_lhs()
    market.link_units_to_generic_constraints(unit_generic_lhs)
    interconnector_generic_lhs = constraint_inputs.get_interconnector_lhs()
    market.link_interconnectors_to_generic_constraints(
        interconnector_generic_lhs)

    # Set the operational demand to be met by dispatch.
    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)

    # Set tiebreak constraint to equalise dispatch of equally priced bids.
    cost = constraint_inputs.get_constraint_violation_prices()['tiebreak']
    market.set_tie_break_constraints(cost)

    # Get unit dispatch without fast start constraints and use it to
    # make fast start unit commitment decisions.
    market.dispatch()
    dispatch = market.get_unit_dispatch()

```

(continues on next page)

(continued from previous page)

```
    fast_start_profiles = unit_inputs.get_fast_start_profiles_for_dispatch(dispatch)
    market.set_fast_start_constraints(fast_start_profiles)

    ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch(run_type="fast_
↪start_second_run")
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.make_constraints_elastic('ramp_up', violation_cost=cost)
    market.make_constraints_elastic('ramp_down', violation_cost=cost)

    cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
    scada_ramp_down_rates = unit_inputs.get_scada_ramp_down_rates_of_lower_reg_units(run_
↪type="fast_start_second_run")
    market.set_joint_ramping_constraints_lower_reg(scada_ramp_down_rates)
    market.make_constraints_elastic('joint_ramping_lower_reg', cost)
    scada_ramp_up_rates = unit_inputs.get_scada_ramp_up_rates_of_raise_reg_units(run_
↪type="fast_start_second_run")
    market.set_joint_ramping_constraints_raise_reg(scada_ramp_up_rates)
    market.make_constraints_elastic('joint_ramping_raise_reg', cost)

    if 'fast_start' in market.get_constraint_set_names.keys():
        cost = constraint_inputs.get_constraint_violation_prices()['fast_start']
        market.make_constraints_elastic('fast_start', violation_cost=cost)

    # If AEMO historically used the over constrained dispatch rerun
    # process then allow it to be used in dispatch. This is needed
    # because sometimes the conditions for over constrained dispatch
    # are present but the rerun process isn't used.
    if constraint_inputs.is_over_constrained_dispatch_rerun():
        market.dispatch(allow_over_constrained_dispatch_re_run=True,
                        energy_market_floor_price=-1000.0,
                        energy_market_ceiling_price=12900.0,
                        fcas_market_ceiling_price=1000.0)
    else:
        # The market price ceiling and floor are not needed here
        # because they are only used for the over constrained
        # dispatch rerun process.
        market.dispatch(allow_over_constrained_dispatch_re_run=False)

    # Save prices from this interval
    prices = market.get_energy_prices()
    prices['time'] = interval

    # Getting historical prices for comparison. Note, ROP price, which is
    # the regional reference node price before the application of any
    # price scaling by AEMO, is used for comparison.
    historical_prices = mms_db_manager.DISPATCHPRICE.get_data(interval)

    prices = pd.merge(prices, historical_prices,
```

(continues on next page)

(continued from previous page)

```
                        left_on=['time', 'region'],
                        right_on=['SETTLEMENTDATE', 'REGIONID'])

    outputs.append(prices.loc[:, ['time', 'region', 'price', 'ROP']])

con.close()

outputs = pd.concat(outputs)

outputs['error'] = outputs['price'] - outputs['ROP']

outputs.to_csv('prices.csv')

print('\n Summary of error in energy price volume weighted average price. \n'
      'Comparison is against ROP, the price prior to \n'
      'any post dispatch adjustments, scaling, capping etc.')
print('Mean price error: {}'.format(outputs['error'].mean()))
print('Median price error: {}'.format(outputs['error'].quantile(0.5)))
print('5% percentile price error: {}'.format(outputs['error'].quantile(0.05)))
print('95% percentile price error: {}'.format(outputs['error'].quantile(0.95)))

#  Summary of error in energy price volume weighted average price.
# Comparison is against ROP, the price prior to
# any post dispatch adjustments, scaling, capping etc.
# Mean price error: 0.0033739383009633033
# Median price error: 0.0
# 5% percentile price error: -0.0001818093963400712
# 95% percentile price error: 0.00981095203372071
```

CHAPTER

FOUR

MARKETS MODULE

A model of the NEM spot market dispatch process.

4.1 Overview

The market, both in real life and in this model, is implemented as a linear program. Linear programs consist of three elements:

1. **Decision variables**: the quantities being optimised for. In an electricity market these will be things like the outputs of generators, the consumption of dispatchable loads and interconnector flows.
2. An **objective function**: the linear function being optimised. In this model of the spot market the cost of production is being minimised, and is defined as the sum of each bids dispatch level multiplied by the bid price.
3. A set of **linear constraints**: used to implement market features such as network constraints and interconnectors.

The class nempy.SpotMarket is used to construct these elements and then solve the linear program to calculate dispatch and pricing. The examples below give an overview of how method calls build the linear program.

- Initialising the market instance, doesn't create any part of the linear program, just saves general information for later use.

```
market = markets.SpotMarket(unit_info=unit_info, market_regions=['NSW'])
```

- Providing volume bids creates a set of n decision variables, where n is the number of bids with a volume greater than zero.

```
market.set_unit_volume_bids(volume_bids)
```

- Providing price bids creates the objective function, i.e. units will be dispatch to minimise cost, as determined by the bid prices.

```
market.set_unit_price_bids(price_bids)
```

- Providing unit capacities creates a constraint for each unit that caps its total dispatch at a set capacity

```
market.set_unit_bid_capacity_constraints(unit_limits)
```

- Providing regional energy demand creates a constraint for each region that forces supply from units and interconnectors to equal demand

```
market.set_demand_constraints(demand)
```

Specific examples for using this class are provided on the **`examples1`_** page, detailed documentation of the class *nempy.markets.SpotMarket* is provided in the *Reference* material below.

4.2 Reference

Classes:

SpotMarket(market_regions, unit_info[, ...])	Class for constructing and dispatching the spot market on an interval basis.

Exceptions:

ModelBuildError	Raise for building model components in wrong order.
MissingTable	Raise for trying to access missing table.

class nempy.markets.**SpotMarket**(*market_regions*, *unit_info*, *dispatch_interval=5*)

Class for constructing and dispatching the spot market on an interval basis.

Examples

Define the unit information data needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                      unit_info=unit_info)
```

The units are given a default dispatch_type and loss_factor. Note this data is stored in a private method and not intended for public use.

```
>>> market._unit_info
  unit region dispatch_type  loss_factor
0    A    NSW     generator          1.0
1    B    NSW     generator          1.0
```

Parameters

- **market_regions** (*list[str]*) – The market regions, used to validate inputs.
- **unit_info** (*pd.DataFrame*) – Information on a unit basis, not all columns are required.

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
region	location of unit, required (as *str*)
loss_factor	marginal, average or combined loss factors, `see AEMO doc`, optional, (as *np.int64*)
dis-patch_type	"load" or "generator", optional, (as *str*)

- **dispatch_interval** (*int*) – The length of the dispatch interval in minutes, used for interpreting ramp rates.

solver_name

The solver to use must be one of solver options of the mip-python package that is used to interface to solvers. Currently the only support solvers are CBC and Gurobi, so allowed solver names are 'CBC' and 'GUROBI'. Default value is CBC, CBC works out of the box after installing Nempy, but Gurobi must be installed separately.

Type
str

Raises

- **RepeatedRowError** – If there is more than one row for any 'unit'.
- **ColumnDataTypeError** – If columns are not of the require type.
- **MissingColumnError** – If the column 'units' or 'regions' is missing.
- **UnexpectedColumn** – There is a column that is not 'units', 'regions', 'dispatch_type' or 'loss_factor'.
- **ColumnValues** – If there are inf, null or negative values in the 'loss_factor' column.

Methods:

set_unit_volume_bids(volume_bids)	Creates the decision variables corresponding to unit bids.
set_unit_price_bids(price_bids)	Creates the objective function costs corresponding to energy bids.
set_unit_bid_capacity_constraints(unit_limits)	Creates constraints that limit unit output based on their bid in max capacity.
set_unconstrained_intermitent_generation_forecast_constraint(...)	Creates constraints that limit unit output based on their forecast output.
set_unit_ramp_up_constraints(ramp_details)	Creates constraints on unit output based on ramp up rate.
set_unit_ramp_down_constraints(ramp_details)	Creates constraints on unit output based on ramp down rate.
set_fast_start_constraints(fast_start_profiles)	Create the constraints on fast start units dispatch, `see AEMO doc`
set_demand_constraints(demand)	Creates constraints that force supply to equal to demand.
set_fcas_requirements_constraints(...)	Creates constraints that force FCAS supply to equal requirements.
set_fcas_max_availability(fcas_max_availability)	Creates constraints to ensure fcas dispatch is limited to the availability specified in the FCAS trapezium.
set_joint_ramping_constraints_raise_reg(...)	Create constraints that ensure the provision of energy and fcas raise are within unit ramping capabilities.
set_joint_ramping_constraints_lower_reg(...)	Create constraints that ensure the provision of energy and fcas are within unit ramping capabilities.
set_joint_capacity_constraints(...)	Creates constraints to ensure there is adequate capacity for contingency, regulation and energy dispatch.
set_energy_and_regulation_capacity_constraints(...)	Creates constraints to ensure there is adequate capacity for regulation and energy dispatch targets.
set_interconnectors(...)	Create lossless links between specified regions.
set_interconnector_losses(loss_functions, ...)	Creates linearised loss functions for interconnectors.
set_generic_constraints(...)	Creates a set of generic constraints, adding the constraint type, rhs.
link_units_to_generic_constraints(...)	Set the lhs coefficients of generic constraints on unit basis.
link_regions_to_generic_constraints(...)	Set the lhs coefficients of generic constraints on region basis.
link_interconnectors_to_generic_constraints(...)	Set the lhs coefficients of generic constraints on an interconnector basis.
make_constraints_elastic(constraints_key, ...)	Make a set of constraints elastic, so they can be violated at a predefined cost.
set_tie_break_constraints(cost)	Creates a cost that attempts to balance the energy dispatch of equally priced bids within a region.
dispatch([energy_market_ceiling_price, ...])	Combines the elements of the linear program and solves to find optimal dispatch.
get_unit_dispatch()	Retrieves the energy dispatch for each unit.
get_energy_prices()	Retrieves the energy price in each market region.
get_fcas_prices()	Retrives the price associated with each set of FCAS requirement constraints.
get_interconnector_flows()	Retrieves the flows for each interconnector.
get_region_dispatch_summary()	Calculates a dispatch summary at the regional level.
get_fcas_availability()	Get the availability of fcas service on a unit level, after constraints.

set_unit_volume_bids(*volume_bids*)

Creates the decision variables corresponding to unit bids.

Variables are created by reserving a variable id (as *int*) for each bid. Bids with a volume of 0 MW do not have a variable created. The lower bound of the variables are set to zero and the upper bound to the bid volume, the variable type is set to continuous. If a no services is specified for the bids they are given the default service value of energy is used.

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 0.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

The market should now have the variables.

```
>>> print(market._decision_variables['bids'])
  unit capacity_band service  variable_id  lower_bound  upper_bound        type
0    A             1  energy            0          0.0         20.0  continuous
1    A             2  energy            1          0.0         20.0  continuous
2    A             3  energy            2          0.0          5.0  continuous
3    B             1  energy            3          0.0         50.0  continuous
4    B             2  energy            4          0.0         30.0  continuous
```

A mapping of these variables to constraints acting on that unit and service should also exist.

```
>>> print(market._variable_to_constraint_map['unit_level']['bids'])
   variable_id unit  service  coefficient
0            0    A   energy          1.0
1            1    A   energy          1.0
2            2    A   energy          1.0
3            3    B   energy          1.0
4            4    B   energy          1.0
```

A mapping of these variables to constraints acting on the units region and service should also exist.

```
>>> print(market._variable_to_constraint_map['regional']['bids'])
   variable_id region service  coefficient
0            0    NSW  energy          1.0
1            1    NSW  energy          1.0
2            2    NSW  energy          1.0
3            3    NSW  energy          1.0
4            4    NSW  energy          1.0
```

Parameters

`volume_bids` (*pd.DataFrame*) – Bids by unit, in MW, can contain up to 10 bid bands, these should be labeled '1' to '10'.

Columns:	Description:
unit	unique identifier of a dispatch unit (as *str*)
service	the service being provided, optional, default 'energy', (as *str*)
1	bid volume in the 1st band, in MW, (as *np.float64*)
2	bid volume in the 2nd band, in MW, optional, (as *np.float64*)
:	
10	bid volume in the nth band, in MW, optional, (as *np.float64*)

Return type

None

Raises

- `RepeatedRowError` – If there is more than one row for any unit and service combination.
- `ColumnDataTypeError` – If columns are not of the require type.
- `MissingColumnError` – If the column 'units' is missing or there are no bid bands.
- `UnexpectedColumn` – There is a column that is not 'unit', 'service' or '1' to '10'.
- `ColumnValues` – If there are inf, null or negative values in the bid band columns.

`set_unit_price_bids`(*price_bids*)

Creates the objective function costs corresponding to energy bids.

If no loss factors have been provided as part of the unit information when the model was initialised then the costs in the objective function are as bid. If loss factors are provided then the bid costs are referred to the regional reference node by dividing by the loss factor.

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids. Bids for each unit need to be monotonically increasing.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [50.0, 100.0],
...     '2': [100.0, 130.0],
...     '3': [100.0, 150.0]})
```

Create the objective function components corresponding to the energy bids.

```
>>> market.set_unit_price_bids(price_bids)
```

The variable assocaited with each bid should now have a cost.

```
>>> print(market._objective_function_components['bids'])
   variable_id unit service capacity_band   cost
0            0    A  energy             1   50.0
1            1    A  energy             2  100.0
2            2    A  energy             3  100.0
3            3    B  energy             1  100.0
4            4    B  energy             2  130.0
5            5    B  energy             3  150.0
```

Parameters

price_bids (*pd.DataFrame*) – Bids by unit, in $/MW, can contain up to 10 bid bands.

Columns:	Description:
unit	unique identifier of a dispatch unit (as *str*)
service	the service being provided, optional, default 'energy', (as *str*)
1	bid price in the 1st band, in $/MW, (as *np.float64*)
2	bid price in the 2nd band, in $/MW, optional, (as *np.float64*)
:	
10	bid price in the nth band, in $/MW, optional, (as *np.float64*)

Return type
None

Raises

- *ModelBuildError* – If the volume bids have not been set yet.
- **RepeatedRowError** – If there is more than one row for any unit and service combination.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the column 'units' is missing or there are no bid bands.
- **UnexpectedColumn** – There is a column that is not 'units', 'region' or '1' to '10'.
- **ColumnValues** – If there are inf, -inf or null values in the bid band columns.
- **BidsNotMonotonicIncreasing** – If the bids band price for all units are not monotonic increasing.

set_unit_bid_capacity_constraints(*unit_limits*)

Creates constraints that limit unit output based on their bid in max capacity. If a unit bids in zero volume then a constraint is not created.

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of unit capacities.

```
>>> unit_limits = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'capacity': [60.0, 100.0]})
```

Create unit capacity based constraints.

```
>>> market.set_unit_bid_capacity_constraints(unit_limits)
```

The market should now have a set of constraints.

```
>>> print(market._constraints_rhs_and_type['unit_bid_capacity'])
  unit service  constraint_id type    rhs
0    A  energy              0   <=   60.0
1    B  energy              1   <=  100.0
```

… and a mapping of those constraints to the variable types on the lhs.

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['unit_bid_capacity'])
   constraint_id unit service  coefficient
0              0    A  energy          1.0
1              1    B  energy          1.0
```

Parameters

unit_limits (*pd.DataFrame*) – Capacity by unit.

Columns:	Description:
unit	unique identifier of a dispatch unit (as *str*)
capacity	The maximum output of the unit if unconstrained by ramp rate, in MW (as *np.float64*)

Return type

None

Raises

- ***ModelBuildError*** – If the volume bids have not been set yet.
- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the required types.
- **MissingColumnError** – If the column 'units' or 'capacity' is missing.
- **UnexpectedColumn** – There is a column that is not 'units' or 'capacity'.
- **ColumnValues** – If there are inf, null or negative values in the bid band columns.

set_unconstrained_intermitent_generation_forecast_constraint(*unit_limits*)

Creates constraints that limit unit output based on their forecast output.

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of unit forecast capacities.

```
>>> unit_limits = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'capacity': [60.0, 100.0]})
```

Create unit capacity based constraints.

```
>>> market.set_unconstrained_intermitent_generation_forecast_constraint(unit_
↪limits)
```

The market should now have a set of constraints.

```
>>> print(market._constraints_rhs_and_type['uigf_capacity'])
  unit service  constraint_id type    rhs
0    A  energy              0   <=   60.0
1    B  energy              1   <=  100.0
```

… and a mapping of those constraints to the variable types on the lhs.

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['uigf_capacity'])
   constraint_id unit service  coefficient
0              0    A  energy          1.0
1              1    B  energy          1.0
```

Parameters

unit_limits (*pd.DataFrame*) – Capacity by unit.

Columns:	Description:
unit	unique identifier of a dispatch unit (as *str*)
capacity	The maximum output of the unit if unconstrained by ramp rate, in MW (as *np.float64*)

Return type
None

Raises

- *ModelBuildError* – If the volume bids have not been set yet.
- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the require type.
- **MissingColumnError** – If the column 'units' or 'capacity' is missing.
- **UnexpectedColumn** – There is a column that is not 'units' or 'capacity'.
- **ColumnValues** – If there are inf, null or negative values in the bid band columns.

set_unit_ramp_up_constraints(*ramp_details*)

Creates constraints on unit output based on ramp up rate.

Constrains the unit output to be <= initial_output + ramp_up_rate * (dispatch_interval / 60)

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info,
...                     dispatch_interval=30)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of unit ramp up rates.

```
>>> ramp_details = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'initial_output': [20.0, 50.0],
...     'ramp_up_rate': [30.0, 100.0]})
```

Create unit capacity based constraints.

```
>>> market.set_unit_ramp_up_constraints(ramp_details)
```

The market should now have a set of constraints.

```
>>> print(market._constraints_rhs_and_type['ramp_up'])
  unit service  constraint_id type    rhs
0    A  energy              0   <=   35.0
1    B  energy              1   <=  100.0
```

… and a mapping of those constraints to variable type for the lhs.

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['ramp_up'])
   constraint_id unit service  coefficient
0              0    A  energy          1.0
1              1    B  energy          1.0
```

Parameters

ramp_details (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
initial_output	the output of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_up_rate	the maximum rate at which the unit can increase output, in MW/h, (as *np.float64*)

Return type

None

Raises

- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the require type.
- **MissingColumnError** – If the column 'units', 'initial_output' or 'ramp_up_rate' is missing.
- **UnexpectedColumn** – There is a column that is not 'units', 'initial_output' or 'ramp_up_rate'.
- **ColumnValues** – If there are inf, null or negative values in the bid band columns.

set_unit_ramp_down_constraints(*ramp_details*)

Creates constraints on unit output based on ramp down rate.

Will constrain the unit output to be >= initial_output - ramp_down_rate * (dispatch_interval / 60).

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info,
...                     dispatch_interval=30)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of unit ramp down rates, also need to provide the initial output of the units at the start of dispatch interval.

```
>>> ramp_details = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'initial_output': [20.0, 50.0],
...     'ramp_down_rate': [20.0, 10.0]})
```

Create unit capacity based constraints.

```
>>> market.set_unit_ramp_down_constraints(ramp_details)
```

The market should now have a set of constraints.

```
>>> print(market._constraints_rhs_and_type['ramp_down'])
  unit service  constraint_id type   rhs
0    A  energy              0   >=  10.0
1    B  energy              1   >=  45.0
```

… and a mapping of those constraints to variable type for the lhs.

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['ramp_down'])
   constraint_id unit service  coefficient
0              0    A  energy          1.0
1              1    B  energy          1.0
```

Parameters
`ramp_details` (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
initial_output	the output of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_down_rate	the maximum rate at which the unit can, decrease output, in MW/h, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the require type.
- **MissingColumnError** – If the column 'units', 'initial_output' or 'ramp_down_rate' is missing.
- **UnexpectedColumn** – There is a column that is not 'units', 'initial_output' or 'ramp_down_rate'.
- **ColumnValues** – If there are inf, null or negative values in the bid band columns.

set_fast_start_constraints(*fast_start_profiles*)

Create the constraints on fast start units dispatch, `see AEMO doc`

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B', 'C', 'D', 'E'],
...     'region': ['NSW', 'NSW', 'NSW', 'NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info,
...                     dispatch_interval=30)
```

Define some example fast start conditions.

```
>>> fast_start_conditions = pd.DataFrame({
...     'unit': ['A', 'B', 'C', 'D', 'E'],
...     'end_mode': [0, 1, 2, 3, 4],
...     'time_in_end_mode': [4.0, 5.0, 5.0, 12.0, 10.0],
...     'mode_two_length': [7.0, 4.0, 10.0, 8.0, 6.0],
...     'mode_four_length': [10.0, 10.0, 20.0, 8.0, 20.0],
...     'min_loading': [30.0, 40.0, 35.0, 50.0, 60.0]})
```

Add fast start constraints.

```
>>> market.set_fast_start_constraints(fast_start_conditions)
```

The market should now have a set of constraints.

```
>>> print(market._constraints_rhs_and_type['fast_start'])
  unit service  constraint_id type   rhs
0    A  energy              0   <=   0.0
1    B  energy              1   <=   0.0
0    C  energy              2   >=  17.5
0    C  energy              3   <=  17.5
0    D  energy              4   >=  50.0
0    E  energy              5   >=  30.0
```

… and a mapping of those constraints to variable type for the lhs.

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['fast_start'])
   constraint_id unit service  coefficient
0              0    A  energy          1.0
1              1    B  energy          1.0
0              3    C  energy          1.0
0              2    C  energy          1.0
0              4    D  energy          1.0
0              5    E  energy          1.0
```

Parameters

fast_start_profiles (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
end_mode	the fast start dispatch mode the unit will end the dispatch interval in, in minutes, (as *np.int64*),
time_in_end_mode	the time the unit will have spent in the end mode at the end of this dispatch interval, in minutes (as *np.int64*)
mode_two_length	the length of dispatch mode 2 for the unit, in minutes, (as *np.int64*)
mode_four_length	the length of dispatch mode 4 for the unit, in minutes, (as *np.int64*)
min_loading	the minimum stable operating level of unit, in MW, (as *np.float64*)

Raises

- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the require type.
- **MissingColumnError** – If any columns are missing.
- **UnexpectedColumn** – If any additional columns are present.
- **ColumnValues** – If there are inf, null or negative values in any of the numeric columns.

set_demand_constraints(*demand*)

Creates constraints that force supply to equal to demand.

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a demand level in each region.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW'],
...     'demand': [100.0]})
```

Create constraints.

```
>>> market.set_demand_constraints(demand)
```

The market should now have a set of constraints.

```
>>> print(market._market_constraints_rhs_and_type['demand'])
  region  constraint_id type    rhs
0    NSW              0    =  100.0
```

. . . and a mapping of those constraints to variable type for the lhs.

```
>>> regional_mapping = market._constraint_to_variable_map['regional']
```

```
>>> print(regional_mapping['demand'])
   constraint_id region service  coefficient
0              0    NSW  energy          1.0
```

Parameters
demand (*pd.DataFrame*) – Demand by region.

Columns:	Description:
region	unique identifier of a region, (as *str*)
demand	the non dispatchable demand, in MW, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the column 'region' or 'demand' is missing.
- **UnexpectedColumn** – There is a column that is not 'region' or 'demand'.
- **ColumnValues** – If there are inf, null or negative values in the volume column.

set_fcas_requirements_constraints(*fcas_requirements*)
Creates constraints that force FCAS supply to equal requirements.

Examples

Define the unit information data set needed to initialise the market, in this example all units are in the same region.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['QLD', 'NSW', 'VIC', 'SA'],
...                     unit_info=unit_info)
```

Define a regulation raise FCAS requirement that apply to all mainland states.

```
>>> fcas_requirements = pd.DataFrame({
...     'set': ['raise_reg_main', 'raise_reg_main',
...             'raise_reg_main', 'raise_reg_main'],
...     'service': ['raise_reg', 'raise_reg',
...                 'raise_reg', 'raise_reg'],
...     'region': ['QLD', 'NSW', 'VIC', 'SA'],
...     'volume': [100.0, 100.0, 100.0, 100.0]})
```

Create constraints.

```
>>> market.set_fcas_requirements_constraints(fcas_requirements)
```

The market should now have a set of constraints.

```
>>> print(market._market_constraints_rhs_and_type['fcas'])
              set  constraint_id type    rhs
0  raise_reg_main              0    =  100.0
```

… and a mapping of those constraints to variable type for the lhs.

```
>>> regional_mapping =              market._constraint_to_variable_map['regional
↪']
```

```
>>> print(regional_mapping['fcas'])
   constraint_id    service region  coefficient
0              0  raise_reg    QLD          1.0
1              0  raise_reg    NSW          1.0
2              0  raise_reg    VIC          1.0
3              0  raise_reg     SA          1.0
```

Parameters

fcas_requirements (*pd.DataFrame*) – requirement by set and the regions and service the requirement applies to.

Columns:	Description:
set	unique identifier of the requirement set, (as *str*)
service	the service or services the requirement set applies to (as *str*)
region	the regions that can contribute to meeting a requirement, (as *str*)
volume	the amount of service required, in MW, (as *np.float64*)
type	the direction of the constrain '=', '>=' or '<=', optional, a value of '=' is assumed if the column is missing (as *str*)

Return type

None

Raises

- **RepeatedRowError** – If there is more than one row for any set, region and service combination.

- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the column 'set', 'service', 'region', or 'volume' is missing.
- **UnexpectedColumn** – There is a column that is not 'set', 'service', 'region', 'volume' or 'type'.
- **ColumnValues** – If there are inf, null or negative values in the volume column.

set_fcas_max_availability(*fcas_max_availability*)

Creates constraints to ensure fcas dispatch is limited to the availability specified in the FCAS trapezium.

The constraints are described in the `FCAS MODEL IN NEMDE documentation section 2`.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define the FCAS max_availability.

```
>>> fcas_max_availability = pd.DataFrame({
... 'unit': ['A'],
... 'service': ['raise_6s'],
... 'max_availability': [60.0]})
```

Set the joint availability constraints.

```
>>> market.set_fcas_max_availability(fcas_max_availability)
```

TNow the market should have the constraints and their mapping to decision varibales.

```
>>> print(market._constraints_rhs_and_type['fcas_max_availability'])
  unit   service  constraint_id type   rhs
0    A  raise_6s              0   <=  60.0
```

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['fcas_max_availability'])
   constraint_id unit   service  coefficient
0              0    A  raise_6s          1.0
```

Parameters

fcas_max_availability (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
service	the fcas service being offered, (as *str*)
max_availability	the maximum volume of the contingency service, in MW, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit and service combination.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the columns 'unit', 'service' or 'max_availability' is missing.
- **UnexpectedColumn** – If there are columns other than 'unit', 'service' or 'max_availability'.
- **ColumnValues** – If there are inf, null or negative values in the columns of type *np.float64*.

set_joint_ramping_constraints_raise_reg(*ramp_details*)

Create constraints that ensure the provision of energy and fcas raise are within unit ramping capabilities.

The constraints are described in the `FCAS MODEL IN NEMDE documentation section 6.1`.

On a unit basis for generators they take the form of:

Energy dispatch + Regulation raise target <= initial output + ramp up rate * (dispatch_interval / 60)

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info,
...                     dispatch_interval=60)
```

Define unit initial outputs and ramping capabilities.

```
>>> ramp_details = pd.DataFrame({
...   'unit': ['A', 'B'],
...   'initial_output': [100.0, 80.0],
...   'ramp_up_rate': [20.0, 10.0]})
```

Create the joint ramping constraints.

```
>>> market.set_joint_ramping_constraints_raise_reg(ramp_details)
```

Now the market should have the constraints and their mapping to decision varibales.

```
>>> print(market._constraints_rhs_and_type['joint_ramping_raise_reg'])
  unit  constraint_id type   rhs
0    A              0   <=  120.0
1    B              1   <=   90.0
```

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['joint_ramping_raise_reg'])
   constraint_id unit    service  coefficient
0              0    A  raise_reg          1.0
1              1    B  raise_reg          1.0
0              0    A     energy          1.0
1              1    B     energy          1.0
```

Parameters
ramp_details (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
ini- tial_output	the output of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_up_rate	the maximum rate at which the unit can increase output, in MW/h, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit in unit_limits.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the columns 'unit', 'initial_output' or 'ramp_up_rate' are missing from unit_limits.
- **UnexpectedColumn** – If there are columns other than 'unit', 'initial_output' or 'ramp_up_rate' in unit_limits.
- **ColumnValues** – If there are inf, null or negative values in the columns of type *np.float64*.

set_joint_ramping_constraints_lower_reg(*ramp_details*)

Create constraints that ensure the provision of energy and fcas are within unit ramping capabilities.

The constraints are described in the `FCAS MODEL IN NEMDE documentation section 6.1`.

On a unit basis for generators they take the form of:

Energy dispatch - Regulation lower target >= initial output - ramp down rate * (dispatch interval / 60)

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info,
...                     dispatch_interval=60)
```

Define unit initial outputs and ramping capabilities.

```
>>> ramp_details = pd.DataFrame({
...   'unit': ['A', 'B'],
...   'initial_output': [100.0, 80.0],
...   'ramp_down_rate': [15.0, 25.0]})
```

Create the joint ramping constraints.

```
>>> market.set_joint_ramping_constraints_lower_reg(ramp_details)
```

Now the market should have the constraints and their mapping to decision varibales.

```
>>> print(market._constraints_rhs_and_type['joint_ramping_lower_reg'])
  unit  constraint_id type   rhs
0    A              0   >=  85.0
1    B              1   >=  55.0
```

```
>>> print(market._constraint_to_variable_map['unit_level']['joint_ramping_lower_
↪reg'])
   constraint_id unit    service  coefficient
0              0    A  lower_reg         -1.0
1              1    B  lower_reg         -1.0
0              0    A     energy          1.0
1              1    B     energy          1.0
```

Parameters
ramp_details (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
initial_output	the output of the unit at the start of, the dispatch interval, in MW, (as *np.float64*)
ramp_down_rate	the maximum rate at which the unit can, decrease output, in MW/h, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit in unit_limits.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the columns 'unit', 'initial_output' or 'ramp_down_rate' are missing from unit_limits.
- **UnexpectedColumn** – If there are columns other than 'unit', 'initial_output' or 'ramp_down_rate' in unit_limits.
- **ColumnValues** – If there are inf, null or negative values in the columns of type *np.float64*.

`set_joint_capacity_constraints`(*contingency_trapeziums*)

Creates constraints to ensure there is adequate capacity for contingency, regulation and energy dispatch.

Create two constraints for each contingency services, one ensures operation on upper slope of the fcas contingency trapezium is consistent with regulation raise and energy dispatch, the second ensures operation on upper slope of the fcas contingency trapezium is consistent with regulation lower and energy dispatch.

The constraints are described in the `FCAS MODEL IN NEMDE documentation section 6.2`.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A'],
...     'region': ['NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define the FCAS contingency trapeziums.

```
>>> contingency_trapeziums = pd.DataFrame({
... 'unit': ['A'],
... 'service': ['raise_6s'],
... 'max_availability': [60.0],
... 'enablement_min': [20.0],
```

(continues on next page)

(continued from previous page)

```
... 'low_break_point': [40.0],
... 'high_break_point': [60.0],
... 'enablement_max': [80.0]})
```

Set the joint capacity constraints.

```
>>> market.set_joint_capacity_constraints(contingency_trapeziums)
```

TNow the market should have the constraints and their mapping to decision varibales.

```
>>> print(market._constraints_rhs_and_type['joint_capacity'])
  unit    service  constraint_id type   rhs
0    A   raise_6s              0   <=  80.0
0    A   raise_6s              1   >=  20.0
```

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['joint_capacity'])
   constraint_id unit    service  coefficient
0              0    A     energy     1.000000
0              0    A   raise_6s     0.333333
0              0    A  raise_reg     1.000000
0              1    A     energy     1.000000
0              1    A   raise_6s    -0.333333
0              1    A  lower_reg    -1.000000
```

Parameters

contingency_trapeziums (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
service	the contingency service being offered, (as *str*)
max_availability	the maximum volume of the contingency service, in MW, (as *np.float64*)
enablement_min	the energy dispatch level at which the unit can begin to provide the, contingency service, in MW, (as *np.float64*)
low_break_point	the energy dispatch level at which the unit can provide the full contingency service offered, in MW, (as *np.float64*)
high_break_point	the energy dispatch level at which the unit can no longer provide the full contingency service offered, in MW, (as *np.float64*)
enablement_max	the energy dispatch level at which the unit can no longer provide the contingency service, in MW, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit and service combination in contingency_trapeziums.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the columns 'unit', 'service', 'max_availability', 'enablement_min', 'low_break_point', 'high_break_point' or 'enablement_max' from contingency_trapeziums.
- **UnexpectedColumn** – If there are columns other than 'unit', 'service', 'max_availability', 'enablement_min', 'low_break_point', 'high_break_point' or 'enablement_max' in contingency_trapeziums.
- **ColumnValues** – If there are inf, null or negative values in the columns of type *np.float64*.

`set_energy_and_regulation_capacity_constraints`(*regulation_trapeziums*)

Creates constraints to ensure there is adequate capacity for regulation and energy dispatch targets.

Create two constraints for each regulation services, one ensures operation on upper slope of the fcas regulation trapezium is consistent with energy dispatch, the second ensures operation on lower slope of the fcas regulation trapezium is consistent with energy dispatch.

The constraints are described in the `FCAS MODEL IN NEMDE documentation section 6.3`.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A'],
...     'region': ['NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define the FCAS regulation trapeziums.

```
>>> regulation_trapeziums = pd.DataFrame({
... 'unit': ['A'],
... 'service': ['raise_reg'],
... 'max_availability': [60.0],
... 'enablement_min': [20.0],
... 'low_break_point': [40.0],
... 'high_break_point': [60.0],
... 'enablement_max': [80.0]})
```

Set the joint capacity constraints.

```
>>> market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)
```

TNow the market should have the constraints and their mapping to decision varibales.

```
>>> print(market._constraints_rhs_and_type['energy_and_regulation_capacity'])
  unit    service  constraint_id type   rhs
0    A  raise_reg              0   <=  80.0
0    A  raise_reg              1   >=  20.0
```

```
>>> unit_mapping = market._constraint_to_variable_map['unit_level']
```

```
>>> print(unit_mapping['energy_and_regulation_capacity'])
   constraint_id unit    service  coefficient
0              0    A     energy     1.000000
0              0    A  raise_reg     0.333333
0              1    A     energy     1.000000
0              1    A  raise_reg    -0.333333
```

Parameters

regulation_trapeziums (*pd.DataFrame*) – The FCAS trapeziums for the regulation services being offered.

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
service	the regulation service being offered, (as *str*)
max_availability	the maximum volume of the contingency service, in MW, (as *np.float64*)
enablement_min	the energy dispatch level at which the unit can begin to provide the regulation service, in MW, (as *np.float64*)
low_break_point	the energy dispatch level at which the unit can provide the full regulation service offered, in MW, (as *np.float64*)
high_break_point	the energy dispatch level at which the unit can no longer provide the full regulation service offered, in MW, (as *np.float64*)
enablement_max	the energy dispatch level at which the unit can no longer provide any regulation service, in MW, (as *np.float64*)

Return type

None

Raises

- **RepeatedRowError** – If there is more than one row for any unit and service combination in regulation_trapeziums.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the columns 'unit', 'service', 'max_availability', 'enablement_min', 'low_break_point', 'high_break_point' or 'enablement_max' from regula-

tion_trapeziums.

- **UnexpectedColumn** – If there are columns other than 'unit', 'service', 'max_availability', 'enablement_min', 'low_break_point', 'high_break_point' or 'enablement_max' in regulation_trapeziums.
- **ColumnValues** – If there are inf, null or negative values in the columns of type *np.float64*.

set_interconnectors(*interconnector_directions_and_limits*)

Create lossless links between specified regions.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A'],
...     'region': ['NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW', 'VIC'],
...                     unit_info=unit_info)
```

Define a an interconnector between NSW and VIC so generator can A can be used to meet demand in VIC.

```
>>> interconnector = pd.DataFrame({
...     'interconnector': ['inter_one'],
...     'to_region': ['VIC'],
...     'from_region': ['NSW'],
...     'max': [100.0],
...     'min': [-100.0]})
```

Create the interconnector.

```
>>> market.set_interconnectors(interconnector)
```

The market should now have a decision variable defined for each interconnector.

```
>>> print(market._decision_variables['interconnectors'])
  interconnector       link  variable_id  lower_bound  upper_bound        type ␣
↪generic_constraint_factor
0      inter_one  inter_one            0       -100.0        100.0  continuous ␣
↪                        1
```

… and a mapping of those variables to to regional energy constraints.

```
>>> regional = market._variable_to_constraint_map['regional']
```

```
>>> print(regional['interconnectors'])
   variable_id interconnector       link region service  coefficient
0            0      inter_one  inter_one    VIC  energy          1.0
1            0      inter_one  inter_one    NSW  energy         -1.0
```

Parameters
`interconnector_directions_and_limits` (*pd.DataFrame*) –

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
to_region	the region that receives power when flow is in the positive direction, (as *str*)
from_region	the region that power is drawn from when flow is in the positive direction, (as *str*)
max	the maximum power flow in the positive direction, in MW, (as *np.float64*)
min	the maximum power flow in the negative direction, in MW, (as *np.float64*)
from_region_loss_factor	the loss factor at the from region end of the interconnector, refers the the from region end to the regional reference node, optional, assumed to equal 1.0, if the column is not provided, (as *np.float*)
to_region_loss_factor	the loss factor at the to region end of the interconnector, refers the to region end to the regional reference node, optional, assumed equal to 1.0 if the column is not provided, (as *np.float*)

Return type
None

Raises

- `RepeatedRowError` – If there is more than one row for any interconnector.
- `ColumnDataTypeError` – If columns are not of the require type.
- `MissingColumnError` – If any columns are missing.
- `UnexpectedColumn` – If there are any additional columns in the input DataFrame.
- `ColumnValues` – If there are inf, null values in the max and min columns.

`set_interconnector_losses`(*loss_functions*, *interpolation_break_points*)

Creates linearised loss functions for interconnectors.

Creates a loss variable for each interconnector, this variable models losses by adding demand to each region. The losses are proportioned to each region according to the from_region_loss_share. In a region with one interconnector, where the region is the nominal from region, the impact on the demand constraint would be:

generation - interconnector flow - interconnector losses * from_region_loss_share = demand

If the region was the nominal to region, then:

generation + interconnector flow - interconnector losses * (1 - from_region_loss_share) = demand

The loss variable is constrained to be a linear interpolation of the loss function between the two break points either side of to the actual line flow. This is achieved using a type 2 Special ordered set, where each variable is bound between 0 and 1, only 2 variables can be greater than 0 and all variables must sum to 1. The actual loss function is evaluated at each break point, the variables of the special order set are constrained such that their values weight the distance of the actual flow from the break points on either side e.g. If we had 3 break points at -100 MW, 0 MW and 100 MW, three weight variables w1, w2, and w3, and a loss function f, then the constraints would be of the form.

Constrain the weight variables to sum to one:

w1 + w2 + w3 = 1

Constrain the weight variables to give the relative weighting of adjacent breakpoint:

w1 * -100.0 + w2 * 0.0 + w3 * 100.0 = interconnector flow

Constrain the interconnector losses to be the weighted sum of the losses at the adjacent break point:

w1 * f(-100.0) + w2 * f(0.0) + w3 * f(100.0) = interconnector losses

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A'],
...     'region': ['NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW', 'VIC'],
...                     unit_info=unit_info)
```

Create the interconnector, this need to be done before a interconnector losses can be set.

```
>>> interconnectors = pd.DataFrame({
...    'interconnector': ['little_link'],
...    'to_region': ['VIC'],
...    'from_region': ['NSW'],
...    'max': [100.0],
...    'min': [-120.0]})
```

```
>>> market.set_interconnectors(interconnectors)
```

Define the interconnector loss function. In this case losses are always 5 % of line flow.

```
>>> def constant_losses(flow):
...     return abs(flow) * 0.05
```

Define the function on a per interconnector basis. Also details how the losses should be proportioned to the connected regions.

```
>>> loss_functions = pd.DataFrame({
...     'interconnector': ['little_link'],
...     'from_region_loss_share': [0.5],  # losses are shared equally.
...     'loss_function': [constant_losses]})
```

Define The points to linearly interpolate the loss function between. In this example the loss function is linear so only three points are needed, but if a non linear loss function was used then more points would result in a better approximation.

```
>>> interpolation_break_points = pd.DataFrame({
...     'interconnector': ['little_link', 'little_link', 'little_link'],
...     'loss_segment': [1, 2, 3],
...     'break_point': [-120.0, 0.0, 100]})
```

```
>>> market.set_interconnector_losses(loss_functions, interpolation_break_points)
```

The market should now have a decision variable defined for each interconnector's losses.

```
>>> print(market._decision_variables['interconnector_losses'])
  interconnector         link  variable_id  lower_bound  upper_bound        type
0    little_link  little_link            1       -120.0        120.0  continuous
```

… and a mapping of those variables to regional energy constraints.

```
>>> print(market._variable_to_constraint_map['regional']['interconnector_losses
↪'])
   variable_id region service  coefficient
0            1    VIC  energy         -0.5
1            1    NSW  energy         -0.5
```

The market will also have a special ordered set of weight variables for interpolating the loss function between the break points.

```
>>> print(market._decision_variables['interpolation_weights'].loc[:,
...       ['interconnector', 'loss_segment', 'break_point', 'variable_id']])
  interconnector  loss_segment  break_point  variable_id
0    little_link             1       -120.0            2
1    little_link             2          0.0            3
2    little_link             3        100.0            4
```

```
>>> print(market._decision_variables['interpolation_weights'].loc[:,
...       ['variable_id', 'lower_bound', 'upper_bound', 'type']])
   variable_id  lower_bound  upper_bound        type
0            2          0.0          1.0  continuous
1            3          0.0          1.0  continuous
2            4          0.0          1.0  continuous
```

and a set of constraints that implement the interpolation, see above explanation.

```
>>> print(market._constraints_rhs_and_type['interpolation_weights'])
  interconnector         link  constraint_id type  rhs
0    little_link  little_link              0    =  1.0
```

```
>>> print(market._constraints_dynamic_rhs_and_type['link_loss_to_flow'])
  interconnector          link  constraint_id type  rhs_variable_id
0    little_link  little_link              2    =                0
0    little_link  little_link              1    =                1
```

```
>>> print(market._lhs_coefficients['interconnector_losses'])
   variable_id  constraint_id  coefficient
0            2              0          1.0
1            3              0          1.0
2            4              0          1.0
0            2              2       -120.0
1            3              2          0.0
2            4              2        100.0
0            2              1          6.0
1            3              1          0.0
2            4              1          5.0
```

Parameters

- **loss_functions** (*pd.DataFrame*) –

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
from_region_loss_share	The fraction of loss occuring in the from region, 0.0 to 1.0, (as *np.float64*)
loss_function	A function that takes a flow, in MW as a float and returns the losses in MW, (as *callable*)

- **interpolation_break_points** (*pd.DataFrame*) –

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
loss_segment	unique identifier of a loss segment on an interconnector basis, (as *np.float64*)
break_point	points between which the loss function will be linearly interpolated, in MW, (as *np.float64*)

Return type

None

Raises

- ***ModelBuildError*** – If all the interconnectors in the input data have not already been added to the model.
- **RepeatedRowError** – If there is more than one row for any interconnector in loss_functions. Or if there is a repeated break point for an interconnector in interpolation_break_points.

- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If any columns are missing.
- **UnexpectedColumn** – If there are any additional columns in the input DataFrames.
- **ColumnValues** – If there are inf or null values in the numeric columns of either input DataFrames. Or if from_region_loss_share are outside the range of 0.0 to 1.0

set_generic_constraints(*generic_constraint_parameters*)

Creates a set of generic constraints, adding the constraint type, rhs.

This sets a set of arbitrary constraints, but only the type and rhs values. The lhs terms can be added to these constraints using the methods link_units_to_generic_constraints, link_interconnectors_to_generic_constraints and link_regions_to_generic_constraints.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A'],
...     'region': ['NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of generic constraints and add them to the market.

```
>>> generic_constraint_parameters = pd.DataFrame({
...   'set': ['A', 'B'],
...   'type': ['>=', '<='],
...   'rhs': [10.0, -100.0]})
```

```
>>> market.set_generic_constraints(generic_constraint_parameters)
```

Now the market should have a set of generic constraints.

```
>>> print(market._constraints_rhs_and_type['generic'])
  set  constraint_id type    rhs
0   A              0   >=   10.0
1   B              1   <= -100.0
```

Parameters

generic_constraint_parameters (*pd.DataFrame*) –

Columns:	Description:
set	the unique identifier of the constraint set, (as *str*)
type	the direction of the constraint >=, <= or =, (as *str*)
rhs	the right hand side value of the constraint, (as *np.float64*)

Return type
None

Raises

- **RepeatedRowError** – If there is more than one row for any unit.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the column 'set', 'type' or 'rhs' is missing.
- **UnexpectedColumn** – There is a column that is not 'set', 'type' or 'rhs' .
- **ColumnValues** – If there are inf or null values in the rhs column.

link_units_to_generic_constraints(*unit_coefficients*)

Set the lhs coefficients of generic constraints on unit basis.

Notes

These sets also maps to the sets in the fcas market constraints. One potential use of this is prevent specific units from helping to meet fcas constraints by giving them a negative one (-1.0) coefficient using this method for particular fcas markey constraints.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'X', 'Y'],
...     'region': ['NSW', 'NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW', 'VIC'],
...                     unit_info=unit_info)
```

Define unit lhs coefficients for generic constraints.

```
>>> unit_coefficients = pd.DataFrame({
...   'set': ['A', 'A', 'B'],
...   'unit': ['X', 'Y', 'X'],
...   'service': ['energy', 'energy', 'raise_reg'],
...   'coefficient': [1.0, 1.0, -1.0]})
```

```
>>> market.link_units_to_generic_constraints(unit_coefficients)
```

Note all this does is save this information to the market object, linking to specific variable ids and constraint id occurs when the dispatch method is called.

```
>>> print(market._generic_constraint_lhs['unit'])
  set unit    service  coefficient
0   A    X     energy          1.0
1   A    Y     energy          1.0
2   B    X  raise_reg         -1.0
```

Parameters

unit_coefficients (*pd.DataFrame*) –

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
unit	the unit whose variables will be mapped to the lhs, (as *str*)
service	the service whose variables will be mapped to the lhs, (as *str*)
coefficient	the lhs coefficient (as *np.float64*)

Raises

- **RepeatedRowError** – If there is more than one row for any set, unit and service combination.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the column 'set', 'unit', 'serice' or 'coefficient' is missing.
- **UnexpectedColumn** – There is a column that is not 'set', 'unit', 'serice' or 'coefficient'.
- **ColumnValues** – If there are inf or null values in the rhs coefficient.

link_regions_to_generic_constraints(*region_coefficients*)

Set the lhs coefficients of generic constraints on region basis.

This effectively acts as short cut for mapping unit variables to a generic constraint. If a particular service in a particular region is included here then all units in this region will have their variables of this service included on the lhs of this constraint set. If a particular unit needs to be excluded from an otherwise region wide constraint it can be given a coefficient with opposite sign to the region wide sign in the generic unit constraints, the coefficients from the two lhs set will be summed and cancel each other out.

Notes

These sets also map to the sets in the fcas market constraints.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['X', 'X']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['X', 'Y'],
...                     unit_info=unit_info)
```

Define region lhs coefficients for generic constraints.

```
>>> region_coefficients = pd.DataFrame({
...     'set': ['A', 'A', 'B'],
...     'region': ['X', 'Y', 'X'],
...     'service': ['energy', 'energy', 'raise_reg'],
...     'coefficient': [1.0, 1.0, -1.0]})
```

```
>>> market.link_regions_to_generic_constraints(region_coefficients)
```

Note all this does is save this information to the market object, linking to specific variable ids and constraint id occurs when the dispatch method is called.

```
>>> print(market._generic_constraint_lhs['region'])
  set region    service  coefficient
0   A      X     energy          1.0
1   A      Y     energy          1.0
2   B      X  raise_reg         -1.0
```

Parameters

region_coefficients (*pd.DataFrame*) –

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
region	the region whose variables will be mapped to the lhs, (as *str*)
service	the service whose variables will be mapped to the lhs, (as *str*)
coefficient	the lhs coefficient (as *np.float64*)

Raises

- **RepeatedRowError** – If there is more than one row for any set, region and service combination.
- **ColumnDataTypeError** – If columns are not of the required type.
- **MissingColumnError** – If the column 'set', 'region', 'service' or 'coefficient' is missing.
- **UnexpectedColumn** – There is a column that is not 'set', 'region', 'service' or 'coefficient'.
- **ColumnValues** – If there are inf or null values in the rhs coefficient.

link_interconnectors_to_generic_constraints(*interconnector_coefficients*)

Set the lhs coefficients of generic constraints on an interconnector basis.

Notes

These sets also map to the set in the fcas market constraints.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['C', 'D'],
...     'region': ['X', 'X']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['X', 'Y'],
...                     unit_info=unit_info)
```

Define region lhs coefficients for generic constraints. All interconnector variables are for the energy service so no 'service' can be specified.

```
>>> interconnector_coefficients = pd.DataFrame({
...   'set': ['A', 'A', 'B'],
...   'interconnector': ['X', 'Y', 'X'],
...   'coefficient': [1.0, 1.0, -1.0]})
```

```
>>> market.link_interconnectors_to_generic_constraints(interconnector_
↪coefficients)
```

Note all this does is save this information to the market object, linking to specific variable ids and constraint id occurs when the dispatch method is called.

```
>>> print(market._generic_constraint_lhs['interconnectors'])
  set interconnector  coefficient
0   A              X          1.0
1   A              Y          1.0
2   B              X         -1.0
```

Parameters

unit_coefficients (*pd.DataFrame*) –

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
intercon-netor	the interconnetor whose variables will be mapped to the lhs, (as *str*)
coefficient	the lhs coefficient (as *np.float64*)

Raises

- **RepeatedRowError** – If there is more than one row for any set, interconnetor and service combination.
- **ColumnDataTypeError** – If columns are not of the required type.

- **MissingColumnError** – If the column 'set', 'interconnetor' or 'coefficient' is missing.
- **UnexpectedColumn** – There is a column that is not 'set', 'interconnetor' or 'coefficient'.
- **ColumnValues** – If there are inf or null values in the rhs coefficient.

make_constraints_elastic(*constraints_key*, *violation_cost*)

Make a set of constraints elastic, so they can be violated at a predefined cost.

If an int or float is provided as the violation_cost, then this directly sets the cost. If a pd.DataFrame is provided then it must contain the columns 'set' and 'cost', 'set' is used to match the cost to the constraints, sets in the constraints that do not appear in the pd.DataFrame will not be made elastic.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['C', 'D'],
...     'region': ['X', 'X']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['X', 'Y'],
...                     unit_info=unit_info)
```

Define a set of generic constraints and add them to the market.

```
>>> generic_constraint_parameters = pd.DataFrame({
...   'set': ['A', 'B'],
...   'type': ['>=', '<='],
...   'rhs': [10.0, -100.0]})
```

```
>>> market.set_generic_constraints(generic_constraint_parameters)
```

Now the market should have a set of generic constraints.

```
>>> print(market._constraints_rhs_and_type['generic'])
  set  constraint_id type    rhs
0   A              0   >=   10.0
1   B              1   <= -100.0
```

Now these constraints can be made elastic.

```
>>> market.make_constraints_elastic('generic', violation_cost=1000.0)
```

Now the market will contain extra decision variables to capture the cost of violating the constraint.

```
>>> print(market._decision_variables['generic_deficit'])
   variable_id  lower_bound  upper_bound        type
0            0          0.0          inf  continuous
1            1          0.0          inf  continuous
```

```
>>> print(market._objective_function_components['generic_deficit'])
   variable_id    cost
0            0  1000.0
1            1  1000.0
```

These will be mapped to the constraints

```
>>> print(market._lhs_coefficients['generic_deficit'])
   variable_id  constraint_id  coefficient
0            0              0          1.0
1            1              1         -1.0
```

If a pd.DataFrame is provided then we can set cost on a constraint basis.

```
>>> violation_cost = pd.DataFrame({
...   'set': ['A', 'B'],
...   'cost': [1000.0, 2000.0]})
```

```
>>> market.make_constraints_elastic('generic', violation_cost=violation_cost)
```

```
>>> print(market._objective_function_components['generic_deficit'])
   variable_id    cost
0            2  1000.0
1            3  2000.0
```

Note will the variable id get incremented with every use of the method only the latest set of variables are used.

Parameters

- **constraints_key** (*str*) – The key used to reference the constraint set in the dict self.market_constraints_rhs_and_type or self.constraints_rhs_and_type. See the documentation for creating the constraint set to get this key.
- **violation_cost** (*str or float or int or pd.DataFrame*) –

Return type

None

Raises

- **ValueError** – If violation_cost is not str, numeric or pd.DataFrame.
- ***ModelBuildError*** – If the constraint_key provided does not match any existing constraints.
- **MissingColumnError** – If violation_cost is a pd.DataFrame and does not contain the columns set and cost. Or if the constraints to be made elastic do not have the set idenetifier.
- **RepeatedRowError** – If violation_cost is a pd.DataFrame and has more than one row per set.
- **ColumnDataTypeError** – If violation_cost is a pd.DataFrame and the column set is not str and the column cost is not numeric.

set_tie_break_constraints(*cost*)

Creates a cost that attempts to balance the energy dispatch of equally priced bids within a region.

For each pair of bids from different generators in a region which are of the same price a constraint of the following form is created.

B1 * 1/C1 - B2 * 1/C2 + D1 - D2 = 0

Where B1 and B2 are the decision variables of each bid, C1 and C2 are the bid volumes, D1 and D2 are additional variables that have provided cost in the objective function. If a small cost (say 1e-6) is provided then this constraint balances the pro rata output of the bids.

For AEMO documentation of this constraint *see AEMO doc <../../docs/pdfs/Schedule of Constraint Violation Penalty factors.pdf>* section 3 item 47.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['X', 'X']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['X'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [50.0, 100.0],
...     '2': [100.0, 130.0],
...     '3': [110.0, 150.0]})
```

```
>>> market.set_unit_price_bids(price_bids)
```

Creat tie break constraints.

```
>>> market.set_tie_break_constraints(1e-3)
```

This should add set of constraints rhs, type and lhs coefficients

```
>>> market._decision_variables['bids']
  unit capacity_band service  variable_id  lower_bound  upper_bound        type
0    A             1  energy            0          0.0         20.0  continuous
1    A             2  energy            1          0.0         20.0  continuous
```

(continues on next page)

(continued from previous page)

```
2    A            3   energy          2         0.0         5.0   continuous
3    B            1   energy          3         0.0        50.0   continuous
4    B            2   energy          4         0.0        30.0   continuous
5    B            3   energy          5         0.0        10.0   continuous
```

```
>>> market._constraints_rhs_and_type['tie_break']
   constraint_id type  rhs
0              0    =  0.0
```

```
>>> market._lhs_coefficients['tie_break']
   constraint_id  variable_id  coefficient
0              0            1         0.05
0              0            3        -0.02
```

And a set of deficiet variables that allow the constraints to violated at the specified cost.

```
>>> market._lhs_coefficients['tie_break_deficit']
   variable_id  constraint_id  coefficient
0            6              0         -1.0
0            7              0          1.0
```

```
>>> market._objective_function_components['tie_break_deficit']
   variable_id   cost
0            6  0.001
0            7  0.001
```

dispatch(*energy_market_ceiling_price=None*, *energy_market_floor_price=None*, *fcas_market_ceiling_price=None*, *allow_over_constrained_dispatch_re_run=False*)

Combines the elements of the linear program and solves to find optimal dispatch.

If allow_over_constrained_dispatch_re_run is set to True then constraints will be relaxed when market ceiling or floor prices are violated.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
```

(continues on next page)

(continued from previous page)

```
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [50.0, 100.0],
...     '2': [100.0, 130.0],
...     '3': [100.0, 150.0]})
```

Create the objective function components corresponding to the the energy bids.

```
>>> market.set_unit_price_bids(price_bids)
```

Define a demand level in each region.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW'],
...     'demand': [100.0]})
```

Create unit capacity based constraints.

```
>>> market.set_demand_constraints(demand)
```

Call the dispatch method.

```
>>> market.dispatch()
```

Now the market dispatch can be retrieved.

```
>>> print(market.get_unit_dispatch())
  unit service  dispatch
0    A  energy      45.0
1    B  energy      55.0
```

And the market prices can be retrieved.

```
>>> print(market.get_energy_prices())
  region  price
0    NSW  130.0
```

Return type
None

Raises
ModelBuildError – If a model build process is incomplete, i.e. there are energy bids but not energy demand set.

get_unit_dispatch()
Retrieves the energy dispatch for each unit.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [50.0, 100.0],
...     '2': [100.0, 130.0],
...     '3': [100.0, 150.0]})
```

Create the objective function components corresponding to the the energy bids.

```
>>> market.set_unit_price_bids(price_bids)
```

Define a demand level in each region.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW'],
...     'demand': [100.0]})
```

Create unit capacity based constraints.

```
>>> market.set_demand_constraints(demand)
```

Call the dispatch method.

```
>>> market.dispatch()
```

Now the market dispatch can be retrieved.

```
>>> print(market.get_unit_dispatch())
  unit service  dispatch
```

(continues on next page)

(continued from previous page)

```
0     A   energy      45.0
1     B   energy      55.0
```

Return type
pd.DataFrame

Raises
ModelBuildError – If a model build process is incomplete, i.e. there are energy bids but not energy demand set.

`get_energy_prices()`

Retrieves the energy price in each market region.

Energy prices are the shadow prices of the demand constraint in each market region.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have two units called A and B, with three bid bands.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [20.0, 50.0],
...     '2': [20.0, 30.0],
...     '3': [5.0, 10.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A', 'B'],
...     '1': [50.0, 100.0],
...     '2': [100.0, 130.0],
...     '3': [100.0, 150.0]})
```

Create the objective function components corresponding to the the energy bids.

```
>>> market.set_unit_price_bids(price_bids)
```

Define a demand level in each region.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW'],
...     'demand': [100.0]})
```

Create unit capacity based constraints.

```
>>> market.set_demand_constraints(demand)
```

Call the dispatch method.

```
>>> market.dispatch()
```

Now the market prices can be retrieved.

```
>>> print(market.get_energy_prices())
  region  price
0    NSW  130.0
```

Return type
pd.DateFrame

Raises
ModelBuildError – If a model build process is incomplete, i.e. there are energy bids but not energy demand set.

get_fcas_prices()

Retrives the price associated with each set of FCAS requirement constraints.

Return type
pd.DateFrame

get_interconnector_flows()

Retrieves the flows for each interconnector.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW', 'VIC'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have just one unit that can provide 100 MW in NSW.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A'],
...     '1': [100.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A'],
...     '1': [80.0]})
```

Create the objective function components corresponding to the the energy bids.

```
>>> market.set_unit_price_bids(price_bids)
```

Define a demand level in each region, no power is required in NSW and 90.0 MW is required in VIC.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW', 'VIC'],
...     'demand': [0.0, 90.0]})
```

Create unit capacity based constraints.

```
>>> market.set_demand_constraints(demand)
```

Define a an interconnector between NSW and VIC so generator can A can be used to meet demand in VIC.

```
>>> interconnector = pd.DataFrame({
...     'interconnector': ['inter_one'],
...     'to_region': ['VIC'],
...     'from_region': ['NSW'],
...     'max': [100.0],
...     'min': [-100.0]})
```

Create the interconnector.

```
>>> market.set_interconnectors(interconnector)
```

Call the dispatch method.

```
>>> market.dispatch()
```

Now the market dispatch can be retrieved.

```
>>> print(market.get_unit_dispatch())
  unit service  dispatch
0    A  energy      90.0
```

And the interconnector flows can be retrieved.

```
>>> print(market.get_interconnector_flows())
  interconnector       link  flow
0      inter_one  inter_one  90.0
```

Return type
pd.DataFrame

get_region_dispatch_summary()

Calculates a dispatch summary at the regional level.

Examples

Define the unit information data set needed to initialise the market.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

Initialise the market instance.

```
>>> market = SpotMarket(market_regions=['NSW', 'VIC'],
...                     unit_info=unit_info)
```

Define a set of bids, in this example we have just one unit that can provide 100 MW in NSW.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A'],
...     '1': [100.0]})
```

Create energy unit bid decision variables.

```
>>> market.set_unit_volume_bids(volume_bids)
```

Define a set of prices for the bids.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A'],
...     '1': [80.0]})
```

Create the objective function components corresponding to the the energy bids.

```
>>> market.set_unit_price_bids(price_bids)
```

Define a demand level in each region, no power is required in NSW and 90.0 MW is required in VIC.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW', 'VIC'],
...     'demand': [0.0, 90.0]})
```

Create unit capacity based constraints.

```
>>> market.set_demand_constraints(demand)
```

Define a an interconnector between NSW and VIC so generator can A can be used to meet demand in VIC.

```
>>> interconnector = pd.DataFrame({
...     'interconnector': ['inter_one'],
...     'to_region': ['VIC'],
...     'from_region': ['NSW'],
...     'max': [100.0],
...     'min': [-100.0]})
```

Create the interconnector.

```
>>> market.set_interconnectors(interconnector)
```

Define the interconnector loss function. In this case losses are always 5 % of line flow.

```
>>> def constant_losses(flow=None):
...     return abs(flow) * 0.05
```

Define the function on a per interconnector basis. Also details how the losses should be proportioned to the connected regions.

```
>>> loss_functions = pd.DataFrame({
...     'interconnector': ['inter_one'],
...     'from_region_loss_share': [0.5],  # losses are shared equally.
...     'loss_function': [constant_losses]})
```

Define the points to linearly interpolate the loss function between. In this example the loss function is linear so only three points are needed, but if a non linear loss function was used then more points would result in a better approximation.

```
>>> interpolation_break_points = pd.DataFrame({
...     'interconnector': ['inter_one', 'inter_one', 'inter_one'],
...     'loss_segment': [1, 2, 3],
...     'break_point': [-120.0, 0.0, 100]})
```

```
>>> market.set_interconnector_losses(loss_functions, interpolation_break_points)
```

Call the dispatch method.

```
>>> market.dispatch()
```

Now the region dispatch summary can be retreived.

```
>>> print(market.get_region_dispatch_summary())
  region   dispatch     inflow  transmission_losses  interconnector_losses
0    NSW  94.615385 -92.307692                  0.0               2.307692
1    VIC   0.000000  92.307692                  0.0               2.307692
```

Returns

Columns:	Description:
region	unique identifier of a market region, required (as *str*)
dispatch	the net dispatch of units inside a region i.e. generators dispatch minus load dispatch, in MW. (as *np.float64*)
inflow	the net inflow from interconnectors, not including losses, in MW (as *np.float64*)
interconnector_losses	interconnector losses attributed to region, in MW, (as *np.float64*)

Return type
pd.DataFrame

get_fcas_availability()

Get the availability of fcas service on a unit level, after constraints.

Examples

Volume of each bid.

```
>>> volume_bids = pd.DataFrame({
...     'unit': ['A', 'A', 'B', 'B', 'B'],
...     'service': ['energy', 'raise_6s', 'energy',
...                 'raise_6s', 'raise_reg'],
...     '1': [100.0, 10.0, 110.0, 15.0, 15.0]})
```

Price of each bid.

```
>>> price_bids = pd.DataFrame({
...     'unit': ['A', 'A', 'B', 'B', 'B'],
...     'service': ['energy', 'raise_6s', 'energy',
...                 'raise_6s', 'raise_reg'],
...     '1': [50.0, 35.0, 60.0, 20.0, 30.0]})
```

Participant defined operational constraints on FCAS enablement.

```
>>> fcas_trapeziums = pd.DataFrame({
...     'unit': ['B', 'B', 'A'],
...     'service': ['raise_reg', 'raise_6s', 'raise_6s'],
...     'max_availability': [15.0, 15.0, 10.0],
...     'enablement_min': [50.0, 50.0, 70.0],
...     'low_break_point': [65.0, 65.0, 80.0],
...     'high_break_point': [95.0, 95.0, 100.0],
...     'enablement_max': [110.0, 110.0, 110.0]})
```

Unit locations.

```
>>> unit_info = pd.DataFrame({
...     'unit': ['A', 'B'],
...     'region': ['NSW', 'NSW']})
```

The demand in the regions being dispatched.

```
>>> demand = pd.DataFrame({
...     'region': ['NSW'],
...     'demand': [195.0]})
```

FCAS requirement in the regions being dispatched.

```
>>> fcas_requirements = pd.DataFrame({
...     'set': ['nsw_regulation_requirement',
...             'nsw_raise_6s_requirement'],
...     'region': ['NSW', 'NSW'],
```

(continues on next page)

(continued from previous page)

```
...     'service': ['raise_reg', 'raise_6s'],
...     'volume': [10.0, 10.0]})
```

Create the market model with unit service bids.

```
>>> market = SpotMarket(unit_info=unit_info,
...                     market_regions=['NSW'])
>>> market.set_unit_volume_bids(volume_bids)
>>> market.set_unit_price_bids(price_bids)
```

Create constraints that enforce the top of the FCAS trapezium.

```
>>> fcas_availability = fcas_trapeziums.loc[:, ['unit', 'service', 'max_
↪availability']]
>>> market.set_fcas_max_availability(fcas_availability)
```

Create constraints the enforce the lower and upper slope of the FCAS regulation service trapeziums.

```
>>> regulation_trapeziums = fcas_trapeziums[fcas_trapeziums['service'] ==
↪'raise_reg']
>>> market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)
```

Create constraints that enforce the lower and upper slope of the FCAS contingency trapezium. These constrains also scale slopes of the trapezium to ensure the co-dispatch of contingency and regulation services is technically feasible.

```
>>> contingency_trapeziums = fcas_trapeziums[fcas_trapeziums['service'] ==
↪'raise_6s']
>>> market.set_joint_capacity_constraints(contingency_trapeziums)
```

Set the demand for energy.

```
>>> market.set_demand_constraints(demand)
```

Set the required volume of FCAS services.

```
>>> market.set_fcas_requirements_constraints(fcas_requirements)
```

Calculate dispatch and pricing

```
>>> market.dispatch()
```

Return the total dispatch of each unit in MW.

```
>>> print(market.get_unit_dispatch())
  unit    service  dispatch
0    A     energy     100.0
1    A   raise_6s       5.0
2    B     energy      95.0
3    B   raise_6s       5.0
4    B  raise_reg      10.0
```

Return the constrained availability of each units fcas service.

```
>>> print(market.get_fcas_availability())
  unit    service  availability
0    A   raise_6s          10.0
1    B   raise_6s           5.0
2    B  raise_reg          10.0
```

exception nempy.markets.**ModelBuildError**

Raise for building model components in wrong order.

exception nempy.markets.**MissingTable**

Raise for trying to access missing table.

CHAPTER

FIVE

HISTORICAL_INPUTS MODULES

The module provides tools for accessing historical market data and preprocessing for compatibility with the SpotMarket class.

5.1 xml_cache

Classes:

XMLCacheManager(cache_folder)	Class for accessing data stored in AEMO's NEMDE output files.

Exceptions:

MissingDataError	Raise for unable to downloaded data from NEMWeb.

class nempy.historical_inputs.xml_cache.**XMLCacheManager**(*cache_folder*)

Class for accessing data stored in AEMO's NEMDE output files.

Examples

A XMLCacheManager instance is created by providing the path to directory containing the cache of XML files.

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

Parameters

cache_folder (*str*) –

Methods:

populate(start_year, start_month, end_year, ...)	Download data to the cache from the AEMO website.
populate_by_day(start_year, start_month, ...)	Download data to the cache from the AEMO website.
load_interval(interval)	Load the data for particular 5 min dispatch interval into memory.
interval_inputs_in_cache()	Check if the cache contains the data for the loaded interval, primarily for debugging.
get_file_path()	Get the file path to the currently loaded interval.
get_file_name()	Get the filename of the currently loaded interval.
get_unit_initial_conditions()	Get the initial conditions of units at the start of the dispatch interval.
get_unit_fast_start_parameters()	Get the unit fast start dispatch inflexibility parameter values.
get_unit_volume_bids()	Get the unit volume bids
get_unit_price_bids()	Get the unit volume bids
get_UIGF_values()	Get the unit unconstrained intermittent generation forecast.
get_violations()	Get the total volume violation of different constraint sets.
get_constraint_violation_prices()	Get the price of violating different constraint sets.
is_intervention_period()	Check if the interval currently loaded was subject to an intervention.
get_constraint_rhs()	Get generic constraints rhs values.
get_constraint_type()	Get generic constraints type.
get_constraint_region_lhs()	Get generic constraints lhs term regional coefficients.
get_constraint_unit_lhs()	Get generic constraints lhs term unit coefficients.
get_constraint_interconnector_lhs()	Get generic constraints lhs term interconnector coefficients.
get_market_interconnector_link_bid_availability()	Get the bid availability of market interconnectors.
find_intervals_with_violations(limit, ...)	Find the set of dispatch intervals where the non-intervention dispatch runs had constraint violations.
get_service_prices()	Get the energy market and FCAS prices by region.

populate(*start_year*, *start_month*, *end_year*, *end_month*, *verbose=True*)

Download data to the cache from the AEMO website. Data downloaded is inclusive of the start and end month.

populate_by_day(*start_year*, *start_month*, *end_year*, *end_month*, *start_day*, *end_day*, *verbose=True*)

Download data to the cache from the AEMO website. Data downloaded is inclusive of the start and end date.

load_interval(*interval*)

Load the data for particular 5 min dispatch interval into memory.

If the file intervals data is not on disk then an attempt to download it from AEMO's NEMweb portal is made.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

Parameters
interval (*str*) – In the format '%Y/%m/%d %H:%M:%S'

Raises
MissingDataError – If the data for an interval is not in the cache and cannot be downloaded from NEMWeb.

interval_inputs_in_cache()

Check if the cache contains the data for the loaded interval, primarily for debugging.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.interval_inputs_in_cache()
True
```

Return type
bool

get_file_path()

Get the file path to the currently loaded interval.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_file_path()
PosixPath('test_nemde_cache/NEMSPDOutputs_2018123124000.loaded')
```

So the doctest runs on all Operating systems lets also look at the parts of the path.

```
>>> manager.get_file_path().parts
('test_nemde_cache', 'NEMSPDOutputs_2018123124000.loaded')
```

get_file_name()

Get the filename of the currently loaded interval.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_file_name()
'NEMSPDOutputs_2018123124000.loaded'
```

get_unit_initial_conditions()

Get the initial conditions of units at the start of the dispatch interval.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_unit_initial_conditions()
         DUID   INITIALMW  RAMPUPRATE  RAMPDOWNRATE  AGCSTATUS
0      AGLHAL    0.000000         NaN           NaN        0.0
1      AGLSOM    0.000000         NaN           NaN        0.0
2     ANGAST1    0.000000         NaN           NaN        0.0
3       APD01    0.000000         NaN           NaN        0.0
4       ARWF1   54.500000         NaN           NaN        0.0
..        ...         ...         ...           ...        ...
283  YARWUN_1  140.360001         NaN           NaN        0.0
284     YWPS1  366.665833  177.750006    177.750006        1.0
285     YWPS2  374.686066  190.125003    190.125003        1.0
286     YWPS3    0.000000  300.374994    300.374994        0.0
287     YWPS4  368.139252  182.249994    182.249994        1.0

[288 rows x 5 columns]
```

Returns

Columns:	Description:
DUID	unique identifier of a dispatch unit, (as *str*)
INI-TIALMW	the output or consumption of the unit at the start of the interval, in MW, (as *np.int64*),
RAM-PUPRATE	ramp up rate of unit as repoted by the scada system at the start if the interval, in MW/h, (as *np.int64*)
RAMP-DOWN-RATE	ramp down rate of unit as repoted by the scada system at the start if the interval, in MW/h, (as *np.int64*)
AGCSTA-TUS	flag to indicate whether the unit is connected to the AGC system at the start of the interval, 0.0 if not and 1.0 if it is, (as *np.int64*)

Return type
pd.DataFrame

get_unit_fast_start_parameters()

Get the unit fast start dispatch inflexibility parameter values.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_unit_fast_start_parameters()
        DUID  MinLoadingMW  CurrentMode  CurrentModeTime  T1  T2  T3  T4
0     AGLHAL             2            0                0  10   3  10   2
1     AGLSOM            16            0                0  20   2  35   2
2   BARCALDN            12            0                0  14   4   1   4
3   BARRON-1             5            4                1  11   3   1   1
4   BARRON-2             5            4                1  11   3   1   1
..       ...           ...          ...              ...  ..  ..  ..  ..
69     VPGS5            48            0                0   5   3  15   0
70     VPGS6            48            0                0   5   3  15   0
71   W/HOE#1           160            0                0   3   0   0   0
72   W/HOE#2           160            0                0   3   0   0   0
73    YABULU            83            0                0   5   6  42   6

[74 rows x 8 columns]
```

Returns

Columns:	Description:
DUID	unique identifier of a dispatch unit, (as *str*)
MinLoadingMW	`see AEMO doc`, in MW, (as *np.int64*)
CurrentMode	The dispatch mode if the unit at the start of the interval, for mode definitions `see AEMO doc`, (as *np.int64*)
CurrentModeTime	The time already spent in the current mode, in minutes, (as *np.int64*)
T1	The total length of mode 1, in minutes (as *np.int64*)
T2	The total length of mode 2, in minutes (as *np.int64*)
T3	The total length of mode 1, in minutes, (as *np.int64*)
T4	The total length of mode 4, in minutes, (as *np.int64*)

Return type
pd.DataFrame

get_unit_volume_bids()

Get the unit volume bids

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_unit_volume_bids()
        DUID     BIDTYPE  MAXAVAIL  ENABLEMENTMIN  ENABLEMENTMAX ␣
↪LOWBREAKPOINT  HIGHBREAKPOINT  BANDAVAIL1  BANDAVAIL2  BANDAVAIL3  BANDAVAIL4␣
↪ BANDAVAIL5  BANDAVAIL6  BANDAVAIL7  BANDAVAIL8  BANDAVAIL9  BANDAVAIL10 ␣
↪RAMPDOWNRATE  RAMPUPRATE
0      AGLHAL      ENERGY     173.0            0.0            0.0            0.
↪0             0.0         0.0         0.0         0.0         0.0         0.0␣
↪        0.0        60.0         0.0         0.0       160.0         720.0    ␣
↪   720.0
1      AGLSOM      ENERGY     160.0            0.0            0.0            0.
↪0             0.0         0.0         0.0         0.0         0.0         0.0␣
↪        0.0         0.0         0.0         0.0       170.0         480.0    ␣
↪   480.0
2     ANGAST1      ENERGY      43.0            0.0            0.0            0.
↪0             0.0         0.0         0.0         0.0         0.0         0.0␣
↪       50.0         0.0         0.0         0.0        50.0         840.0    ␣
↪   840.0
3       APD01    LOWER5MIN       0.0            0.0            0.0            0.
```

(continues on next page)

(continued from previous page)

```
↪0          0.0          0.0          0.0          0.0          0.0          0.0␣
↪       0.0          0.0          0.0          0.0       300.0          0.0    ␣
↪    0.0
4        APD01  LOWER60SEC        0.0          0.0          0.0          0.
↪0          0.0          0.0          0.0          0.0          0.0          0.0␣
↪       0.0          0.0          0.0          0.0       300.0          0.0    ␣
↪    0.0
...        ...         ...        ...          ...          ...          ...
↪            ...          ...          ...          ...          ...          ... ␣
↪      ...          ...          ...          ...          ...          ...    ␣
↪    ...
1021     YWPS4   LOWER6SEC       25.0        250.0        385.0        275.
↪0        385.0         15.0         10.0          0.0          0.0          0.0␣
↪       0.0          0.0          0.0          0.0          0.0          0.0    ␣
↪    0.0
1022     YWPS4   RAISE5MIN        0.0        250.0        390.0        250.
↪0        380.0          0.0          0.0          0.0          0.0          5.0␣
↪       0.0          0.0          5.0          0.0         10.0          0.0    ␣
↪    0.0
1023     YWPS4    RAISEREG       15.0        250.0        385.0        250.
↪0        370.0          0.0          0.0          0.0          0.0          0.0␣
↪       0.0          5.0         10.0          0.0          5.0          0.0    ␣
↪    0.0
1024     YWPS4  RAISE60SEC       10.0        220.0        400.0        220.
↪0        390.0          0.0          0.0          0.0          0.0          0.0␣
↪       5.0          5.0          0.0          0.0         10.0          0.0    ␣
↪    0.0
1025     YWPS4   RAISE6SEC       15.0        220.0        405.0        220.
↪0        390.0          0.0          0.0          0.0         10.0          5.0␣
↪       0.0          0.0          0.0          0.0         10.0          0.0    ␣
↪    0.0

[1026 rows x 19 columns]
```

Returns

Columns:	Description:
DUID	unique identifier of a dispatch unit, (as *str*)
BIDTYPE	the service the bid applies to, (as *str*)
MAXAVAIL	the bid in unit availablity, in MW, (as *str*)
ENABLEMENTMIN	`see AMEO docs`, in MW, (as *np.float64*)
ENABLEMENTMAX	`see AMEO docs`, in MW, (as *np.float64*)
LOWBREAKPOINT	`see AMEO docs`, in MW, (as *np.float64*)
HIGHBREAKPOINT	`see AMEO docs`, in MW, (as *np.float64*)
BANDAVAIL1	the volume bid in the first bid band, in MW, (as *np.float64*)
:	
BANDAVAIL10	the volume bid in the tenth bid band, in MW, (as *np.float64*)
RAMPDOWNRATE	the bid in ramp down rate, in MW/h, (as *np.int64*)
RAMPUPRATE	the bid in ramp up rate, in MW/h, (as *np.int64*)

Return type
pd.DataFrame

get_unit_price_bids()
Get the unit volume bids

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_unit_price_bids()
         DUID     BIDTYPE  PRICEBAND1  PRICEBAND2  PRICEBAND3  PRICEBAND4 ␣
↪PRICEBAND5  PRICEBAND6  PRICEBAND7  PRICEBAND8  PRICEBAND9  PRICEBAND10
0      AGLHAL      ENERGY    -1000.00        0.00      278.81      368.81    ␣
↪418.81      498.81      578.81     1365.56    10578.87     13998.99
1      AGLSOM      ENERGY    -1000.00        0.00       85.00      110.00    ␣
↪145.00      284.00      451.00     1001.00    13300.87     14499.96
2     ANGAST1      ENERGY    -1000.00        0.00      125.00      200.20    ␣
↪299.19      379.98      589.99     1374.85    10618.00     14500.00
3       APD01   LOWER5MIN        0.00        1.00        2.00        3.00     ␣
↪ 4.00        5.00        6.00        7.00        8.00         9.00
4       APD01  LOWER60SEC        0.00        1.00        2.00        3.00     ␣
↪ 4.00        5.00        6.00        7.00        8.00         9.00
...       ...         ...         ...         ...         ...         ...     ␣
↪  ...         ...         ...         ...         ...          ...
1021    YWPS4   LOWER6SEC        0.03        0.05        0.16        0.30     ␣
↪ 1.90       25.04       30.04       99.00     4600.00      9899.00
1022    YWPS4   RAISE5MIN        0.05        1.78        4.48       14.50     ␣
```

(continues on next page)

(continued from previous page)

```
↪30.03        49.00        87.70       100.00     11990.00      12400.40
1023     YWPS4     RAISEREG         0.05         2.70         9.99        19.99     ␣
↪49.00        95.50       240.00       450.50       950.50      11900.00
1024     YWPS4   RAISE60SEC         0.17         1.80         4.80        10.01     ␣
↪21.00        39.00        52.00       102.00      4400.00      11999.00
1025     YWPS4    RAISE6SEC         0.48         1.75         4.90        20.70     ␣
↪33.33        99.90       630.00      1999.00      6000.00      12299.00

[1026 rows x 12 columns]
```

Returns

Columns:	Description:
DUID	unique identifier of a dispatch unit, (as *str*)
BIDTYPE	the service the bid applies to, (as *str*)
PRICEBAND1	the volume bid in the first bid band, in MW, (as *np.float64*)
:	
PRICEBAND10	the volume bid in the tenth bid band, in MW, (as *np.float64*)

Return type
pd.DataFrame

get_UIGF_values()

Get the unit unconstrained intermittent generation forecast.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_UIGF_values()
        DUID    UIGF
0      ARWF1  56.755
1   BALDHWF1   9.160
2      BANN1   0.000
3     BLUFF1   4.833
4     BNGSF1   0.000
..       ...     ...
57     WGWF1  25.445
58   WHITSF1   0.000
59  WOODLWN1   0.075
60     WRSF1   0.000
61     WRWF1  15.760

[62 rows x 2 columns]
```

Returns

Columns:	Description:
DUID	unique identifier of a dispatch unit, (as *str*)
UGIF	the units generation forecast for end of the inteval, in MW, (as *np.float64*)

Return type
pd.DataFrame

`get_violations()`

Get the total volume violation of different constraint sets.

For more information on the constraint sets `see AMEO docs`

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_violations()
{'regional_demand': 0.0, 'interocnnector': 0.0, 'generic_constraint': 0.0,
↪'ramp_rate': 0.0, 'unit_capacity': 0.36, 'energy_constraint': 0.0, 'energy_
↪offer': 0.0, 'fcas_profile': 0.0, 'fast_start': 0.0, 'mnsp_ramp_rate': 0.0,
↪'msnp_offer': 0.0, 'mnsp_capacity': 0.0, 'ugif': 0.0}
```

Return type
dict

`get_constraint_violation_prices()`

Get the price of violating different constraint sets.

For more information on the constraint sets `see AMEO docs`

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_constraint_violation_prices()
{'regional_demand': 2175000.0, 'interocnnector': 16675000.0, 'generic_constraint
↪': 435000.0, 'ramp_rate': 16747500.0, 'unit_capacity': 5365000.0, 'energy_
↪offer': 16457500.0, 'fcas_profile': 2247500.0, 'fcas_max_avail': 2247500.0,
↪'fcas_enablement_min': 1015000.0, 'fcas_enablement_max': 1015000.0, 'fast_
↪start': 16385000.0, 'mnsp_ramp_rate': 16747500.0, 'msnp_offer': 16457500.0,
↪'mnsp_capacity': 5292500.0, 'uigf': 5582500.0, 'voll': 14500.0, 'tiebreak':␣
↪1e-06}
```

Return type
dict

is_intervention_period()
Check if the interval currently loaded was subject to an intervention.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.is_intervention_period()
False
```

Return type
bool

get_constraint_rhs()
Get generic constraints rhs values.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_constraint_rhs()
                     set           rhs
0               #BANN1_E     32.000000
1              #BNGSF2_E      3.000000
2            #CROWLWF1_E     43.000000
3             #CSPVPS1_E     29.000000
4             #DAYDSF1_E      0.000000
..                   ...           ...
704          V_OWF_NRB_0  10000.001000
705   V_OWF_TGTSNRBHTN_30  10030.000000
706         V_S_NIL_ROCOF    812.280029
707           V_T_NIL_BL1    478.000000
708         V_T_NIL_FCSPS    425.154024

[709 rows x 2 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the generic constraint, (as *str*)
rhs	the rhs value of the constraint, (as *np.float64*)

Return type
pd.DataFrame

get_constraint_type()

Get generic constraints type.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_constraint_type()
                     set type       cost
0               #BANN1_E   LE  5220000.0
1              #BNGSF2_E   LE  5220000.0
2            #CROWLWF1_E   LE  5220000.0
3             #CSPVPS1_E   LE  5220000.0
4             #DAYDSF1_E   LE  5220000.0
..                   ...  ...        ...
704          V_OWF_NRB_0   LE  5220000.0
705  V_OWF_TGTSNRBHTN_30   LE  5220000.0
706        V_S_NIL_ROCOF   LE   507500.0
707          V_T_NIL_BL1   LE  5220000.0
708        V_T_NIL_FCSPS   LE   435000.0

[709 rows x 3 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the generic constraint, (as *str*)
type	the type of constraint, i.e '=', '<=' or '<=', (as *str*)
cost	the cost of violating the constraint, (as *np.float64*)

Return type
pd.DataFrame

get_constraint_region_lhs()

Get generic constraints lhs term regional coefficients.

This is a compact way of describing constraints that apply to all units in a region. If a constraint set appears here and also in the unit specific lhs table then the coefficents used in the constraint is the sum of the two coefficients, this can be used to exclude particular units from otherwise region wide constraints.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_constraint_region_lhs()
                   set region service  coefficient
0         F_I+LREG_0120   NSW1    L5RE          1.0
1         F_I+LREG_0120   QLD1    L5RE          1.0
2         F_I+LREG_0120    SA1    L5RE          1.0
3         F_I+LREG_0120   TAS1    L5RE          1.0
4         F_I+LREG_0120   VIC1    L5RE          1.0
..                  ...    ...     ...          ...
478   F_T+NIL_WF_TG_R5   TAS1    R5RE          1.0
479   F_T+NIL_WF_TG_R6   TAS1    R6SE          1.0
480  F_T+NIL_WF_TG_R60   TAS1    R60S          1.0
481       F_T+RREG_0050   TAS1    R5RE          1.0
482     F_T_NIL_MINP_R6   TAS1    R6SE          1.0

[483 rows x 4 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the generic constraint, (as *str*)
region	the regions the constraint applies in, (as *str*)
service	the services the constraint applies too, (as *str*)
coefficient	the coefficient of the terms on the lhs, (as *np.float64*)

Return type

pd.DataFrame

get_constraint_unit_lhs()

Get generic constraints lhs term unit coefficients.

If a constraint set appears here and also in the region lhs table then the coefficents used in the constraint is the sum of the two coefficients, this can be used to exclude particular units from otherwise region wide constraints.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_constraint_unit_lhs()
                     set      unit service  coefficient
0              #BANN1_E     BANN1    ENOF          1.0
1             #BNGSF2_E    BNGSF2    ENOF          1.0
2           #CROWLWF1_E  CROWLWF1    ENOF          1.0
3            #CSPVPS1_E   CSPVPS1    ENOF          1.0
4            #DAYDSF1_E   DAYDSF1    ENOF          1.0
...                 ...       ...     ...          ...
5864      V_ARWF_FSTTRP_5     ARWF1    ENOF          1.0
5865      V_MTGBRAND_33WT  MTGELWF1    ENOF          1.0
5866     V_OAKHILL_TFB_42  OAKLAND1    ENOF          1.0
5867           V_OWF_NRB_0  OAKLAND1    ENOF          1.0
5868  V_OWF_TGTSNRBHTN_30  OAKLAND1    ENOF          1.0

[5869 rows x 4 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the generic constraint, (as *str*)
unit	the units the constraint applies in, (as *str*)
service	the services the constraint applies too, (as *str*)
coefficient	the coefficient of the terms on the lhs, (as *np.float64*)

Return type

pd.DataFrame

get_constraint_interconnector_lhs()

Get generic constraints lhs term interconnector coefficients.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_constraint_interconnector_lhs()
                    set interconnector  coefficient
0              DATASNAP      N-Q-MNSP1          1.0
```

(continues on next page)

(continued from previous page)

```
1        DATASNAP_DFS_LS      N-Q-MNSP1          1.0
2      DATASNAP_DFS_NCAN      N-Q-MNSP1          1.0
3    DATASNAP_DFS_NCWEST      N-Q-MNSP1          1.0
4      DATASNAP_DFS_NNTH      N-Q-MNSP1          1.0
..                   ...            ...          ...
631       V^^S_NIL_TBSE_1          V-SA          1.0
632       V^^S_NIL_TBSE_2          V-SA          1.0
633         V_S_NIL_ROCOF          V-SA          1.0
634           V_T_NIL_BL1     T-V-MNSP1         -1.0
635         V_T_NIL_FCSPS     T-V-MNSP1         -1.0

[636 rows x 3 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the generic constraint, (as *str*)
interconnector	the interconnector the constraint applies in, (as *str*)
coefficient	the coefficient of the terms on the lhs, (as *np.float64*)

Return type
pd.DataFrame

get_market_interconnector_link_bid_availability()

Get the bid availability of market interconnectors.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_market_interconnector_link_bid_availability()
  interconnector to_region  availability
0      T-V-MNSP1      TAS1         478.0
1      T-V-MNSP1      VIC1         478.0
```

Returns

Columns:	Description:
interconnector	the interconnector the constraint applies in, (as *str*)
to_region	the direction the bid availability applies to, (as *str*)
availability	the availability as bid in by the interconnector, (as *str*)

Return type
pd.DataFrame

find_intervals_with_violations(*limit*, *start_year*, *start_month*, *end_year*, *end_month*)

Find the set of dispatch intervals where the non-intervention dispatch runs had constraint violations.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.find_intervals_with_violations(limit=3, start_year=2019, start_
↪month=1, end_year=2019, end_month=1)
{'2019/01/01 00:00:00': ['unit_capacity'], '2019/01/01 00:05:00': ['unit_
↪capacity'], '2019/01/01 00:10:00': ['unit_capacity']}
```

Parameters

- **limit** (*int*) – number of intervals to find, finds first intervals in chronolgical order
- **start_year** (*int*) – year to start search
- **start_month** (*int*) – month to start search
- **end_year** (*int*) – year to end search
- **end_month** (*int*) – month to end search

Return type
dict

get_service_prices()

Get the energy market and FCAS prices by region.

Examples

```
>>> manager = XMLCacheManager('test_nemde_cache')
```

```
>>> manager.load_interval('2019/01/01 00:00:00')
```

```
>>> manager.get_service_prices()
   region     service     price
0    NSW1      ENERGY  62.93553
1    NSW1   RAISE5MIN      4.39
2    NSW1  RAISE60SEC         1
3    NSW1  LOWER60SEC      0.07
4    NSW1   RAISE6SEC         1
5    NSW1   LOWER6SEC      0.03
6    QLD1      ENERGY  58.71004
7    QLD1   RAISE5MIN      4.39
8    QLD1  RAISE60SEC         1
```

(continues on next page)

(continued from previous page)

```
9    QLD1  LOWER60SEC      0.07
10   QLD1   RAISE6SEC         1
11   QLD1   LOWER6SEC      0.03
12    SA1      ENERGY   79.0014
13    SA1   RAISE5MIN      4.39
14    SA1  RAISE60SEC         1
15    SA1  LOWER60SEC      0.07
16    SA1   RAISE6SEC         1
17    SA1   LOWER6SEC      0.03
18   TAS1      ENERGY  79.00957
19   TAS1   RAISE5MIN      14.4
20   TAS1  RAISE60SEC      4.95
21   TAS1  LOWER60SEC      0.07
22   TAS1   RAISE6SEC      4.95
23   TAS1   LOWER6SEC      0.03
24   VIC1      ENERGY  75.23031
25   VIC1   RAISE5MIN      4.39
26   VIC1  RAISE60SEC         1
27   VIC1  LOWER60SEC      0.07
28   VIC1   RAISE6SEC         1
29   VIC1   LOWER6SEC      0.03
```

Returns

Columns:	Description:
region	the region (as *str*)
service	the services (as *str*), i.e. energy, lower_1s, lower_5min, etc
price	the price of the service (as *np.float64*)

Return type
pd.DataFrame

`exception nempy.historical_inputs.xml_cache.`**`MissingDataError`**

Raise for unable to downloaded data from NEMWeb.

`with_traceback()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

5.2 mms_db

Classes:

DBManager(connection)	Constructs and manages a sqlite database for accessing historical inputs for NEM spot market dispatch.
InputsBySettlementDate(table_name, ...)	Manages retrieving dispatch inputs by SETTLEMENT-DATE.
InputsByIntervalDateTime(table_name, ...)	Manages retrieving dispatch inputs by INTER-VAL_DATETIME.
InputsByDay(table_name, table_columns, ...)	Manages retrieving dispatch inputs by SETTLEMENT-DATE, where inputs are stored on a daily basis.
InputsStartAndEnd(table_name, table_columns, ...)	Manages retrieving dispatch inputs by START_DATE and END_DATE.
InputsByMatchDispatchConstraints(table_name, ...)	Manages retrieving dispatch inputs by matching against the DISPATCHCONSTRAINTS table
InputsByEffectiveDateVersionNoAndDispatchInterconnector(...)	Manages retrieving dispatch inputs by EFFECTTIVE-DATE and VERSIONNO.
InputsByEffectiveDateVersionNo(table_name, ...)	Manages retrieving dispatch inputs by EFFECTTIVE-DATE and VERSIONNO.
InputsNoFilter(table_name, table_columns, ...)	Manages retrieving dispatch inputs where no filter is re-quire.

class nempy.historical_inputs.mms_db.**DBManager**(*connection*)

Constructs and manages a sqlite database for accessing historical inputs for NEM spot market dispatch.

Constructs a database if none exists, otherwise connects to an existing database. Specific datasets can be added to the database from AEMO nemweb portal and inputs can be retrieved on a 5 min dispatch interval basis.

Examples

Create the database or connect to an existing one.

```
>>> import sqlite3
```

```
>>> con = sqlite3.connect('historical.db')
```

Create the database manager.

```
>>> historical = DBManager(con)
```

Create a set of default table in the database.

```
>>> historical.create_tables()
```

Add data from AEMO nemweb data portal. In this case we are adding data from the table DISPATCHREGION-SUM which contains a dispatch summary by region, the data comes in monthly chunks.

```
>>> historical.DISPATCHREGIONSUM.add_data(year=2020, month=1)
```

```
>>> historical.DISPATCHREGIONSUM.add_data(year=2020, month=2)
```

This table has an add_data method indicating that data provided by AEMO comes in monthly files that do not overlap. If you need data for multiple months then multiple add_data calls can be made.

Data for a specific 5 min dispatch interval can then be retrieved.

```
>>> print(historical.DISPATCHREGIONSUM.get_data('2020/01/10 12:35:00').head())
        SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/01/10 12:35:00     NSW1      9938.01        34.23926     9902.79199
1  2020/01/10 12:35:00     QLD1      6918.63        26.47852     6899.76270
2  2020/01/10 12:35:00      SA1      1568.04         4.79657     1567.85864
3  2020/01/10 12:35:00     TAS1      1124.05        -3.43994     1109.36963
4  2020/01/10 12:35:00     VIC1      6633.45        37.05273     6570.15527
```

Some tables will have a set_data method instead of an add_data method, indicating that the most recent data file provided by AEMO contains all historical data for this table. In this case if multiple calls to the set_data method are made the new data replaces the old.

```
>>> historical.DUDETAILSUMMARY.set_data(year=2020, month=2)
```

Data for a specific 5 min dispatch interval can then be retrieved.

```
>>> print(historical.DUDETAILSUMMARY.get_data('2020/01/10 12:35:00').head())
       DUID           START_DATE             END_DATE DISPATCHTYPE ␣
↪CONNECTIONPOINTID REGIONID  TRANSMISSIONLOSSFACTOR  DISTRIBUTIONLOSSFACTOR ␣
↪SCHEDULE_TYPE
0    AGLHAL  2019/07/01 00:00:00  2020/01/20 00:00:00    GENERATOR             ␣
↪SHPS1       SA1                 0.9748                  1.0000      SCHEDULED
1   AGLNOW1  2019/07/01 00:00:00  2999/12/31 00:00:00    GENERATOR             ␣
↪NDT12      NSW1                 0.9929                  1.0000  NON-SCHEDULED
2  AGLSITA1  2019/07/01 00:00:00  2999/12/31 00:00:00    GENERATOR            ␣
↪NLP13K      NSW1                 1.0009                  1.0000  NON-SCHEDULED
3    AGLSOM  2019/07/01 00:00:00  2999/12/31 00:00:00    GENERATOR             ␣
↪VTTS1      VIC1                 0.9915                  0.9891      SCHEDULED
4   ANGAST1  2019/07/01 00:00:00  2999/12/31 00:00:00    GENERATOR             ␣
↪SDRN1       SA1                 0.9517                  0.9890      SCHEDULED
```

Parameters

con (*sqlite3.connection*) –

BIDPEROFFER_D

Unit volume bids by 5 min dispatch intervals.

Type

InputsByIntervalDateTime

BIDDAYOFFER_D

Unit price bids by market day.

Type

InputsByDay

DISPATCHREGIONSUM

Regional demand terms by 5 min dispatch intervals.

Type

InputsBySettlementDate

DISPATCHLOAD

Unit operating conditions by 5 min dispatch intervals.

Type
InputsBySettlementDate

`DUDETAILSUMMARY`

Unit information by the start and end times of when the information is applicable.

Type
InputsStartAndEnd

`DISPATCHCONSTRAINT`

The generic constraints that were used in each 5 min interval dispatch.

Type
InputsBySettlementDate

`GENCONDATA`

The generic constraints information, their applicability to a particular dispatch interval is determined by reference to DISPATCHCONSTRAINT.

Type
InputsByMatchDispatchConstraints

`SPDREGIONCONSTRAINT`

The regional lhs terms in generic constraints, their applicability to a particular dispatch interval is determined by reference to DISPATCHCONSTRAINT.

Type
InputsByMatchDispatchConstraints

`SPDCONNECTIONPOINTCONSTRAINT`

The connection point lhs terms in generic constraints, their applicability to a particular dispatch interval is determined by reference to DISPATCHCONSTRAINT.

Type
InputsByMatchDispatchConstraints

`SPDINTERCONNECTORCONSTRAINT`

The interconnector lhs terms in generic constraints, their applicability to a particular dispatch interval is determined by reference to DISPATCHCONSTRAINT.

Type
InputsByMatchDispatchConstraints

`INTERCONNECTOR`

The the regions that each interconnector links.

Type
InputsNoFilter

`INTERCONNECTORCONSTRAINT`

Interconnector properties FROMREGIONLOSSSHARE, LOSSCONSTANT, LOSSFLOWCOEFFICIENT, MAXMWIN, MAXMWOUT by EFFECTIVEDATE and VERSIONNO.

Type
InputsByEffectiveDateVersionNoAndDispatchInterconnector

`LOSSMODEL`

Break points used in linearly interpolating interconnector loss funtctions by EFFECTIVEDATE and VERSIONNO.

Type
InputsByEffectiveDateVersionNoAndDispatchInterconnector

LOSSFACTORMODEL
Coefficients of demand terms in interconnector loss functions.

Type
InputsByEffectiveDateVersionNoAndDispatchInterconnector

DISPATCHINTERCONNECTORRES
Record of which interconnector were used in a particular dispatch interval.

Type
InputsBySettlementDate

Methods:

create_tables()	Drops any existing default tables and creates new ones, this method is generally called a new database.

create_tables()
Drops any existing default tables and creates new ones, this method is generally called a new database.

Examples

Create the database or connect to an existing one.

```
>>> import sqlite3
```

```
>>> con = sqlite3.connect('historical.db')
```

Create the database manager.

```
>>> historical = DBManager(con)
```

Create a set of default table in the database.

```
>>> historical.create_tables()
```

Default tables will now exist, but will be empty.

```
>>> print(pd.read_sql("Select * from DISPATCHREGIONSUM", con=con))
Empty DataFrame
Columns: [SETTLEMENTDATE, REGIONID, TOTALDEMAND, DEMANDFORECAST, INITIALSUPPLY]
Index: []
```

If you added data and then call create_tables again then any added data will be emptied.

```
>>> historical.DISPATCHREGIONSUM.add_data(year=2020, month=1)
```

```
>>> print(pd.read_sql("Select * from DISPATCHREGIONSUM limit 3", con=con))
        SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/01/01 00:05:00     NSW1      7245.31       -26.35352     7284.32178
1  2020/01/01 00:05:00     QLD1      6095.75       -24.29639     6129.36279
2  2020/01/01 00:05:00      SA1      1466.53         1.47190     1452.25647
```

```
>>> historical.create_tables()
```

```
>>> print(pd.read_sql("Select * from DISPATCHREGIONSUM", con=con))
Empty DataFrame
Columns: [SETTLEMENTDATE, REGIONID, TOTALDEMAND, DEMANDFORECAST, INITIALSUPPLY]
Index: []
```

Return type
None

class nempy.historical_inputs.mms_db.**InputsBySettlementDate**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs by SETTLEMENTDATE.

Methods:

get_data(date_time)	Retrieves data for the specified date_time e.g.
add_data(year, month)	"Download data for the given table and time, appends to any existing data.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.

get_data(*date_time*)

Retrieves data for the specified date_time e.g. 2019/01/01 11:55:00"

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = InputsBySettlementDate(table_name='EXAMPLE', table_columns=[
↪'SETTLEMENTDATE', 'INITIALMW'],
...                                 table_primary_keys=['SETTLEMENTDATE'],␣
↪con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the add_data method to add historical data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...     'SETTLEMENTDATE': ['2019/01/01 11:55:00', '2019/01/01 12:00:00'],
...     'INITIALMW': [1.0, 2.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

When we call get_data the output is filtered by SETTLEMENTDATE.

```
>>> print(table.get_data(date_time='2019/01/01 12:00:00'))
       SETTLEMENTDATE  INITIALMW
0  2019/01/01 12:00:00        2.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters
date_time (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

add_data(*year*, *month*)

"Download data for the given table and time, appends to any existing data.

Note: This method and its documentation is inherited from the _MultiDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MultiDataSource(table_name='DISPATCHREGIONSUM',
...   table_columns=['SETTLEMENTDATE', 'REGIONID', 'TOTALDEMAND',
...                  'DEMANDFORECAST', 'INITIALSUPPLY'],
...   table_primary_keys=['SETTLEMENTDATE', 'REGIONID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.add_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DISPATCHREGIONSUM order by SETTLEMENTDATE DESC limit␣
↪1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
       SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/02/01 00:00:00     VIC1       5935.1        -15.9751     5961.77002
```

If we subsequently add data from an earlier month the old data remains in the table, in addition to the new data.

```
>>> table.add_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/02/01 00:00:00     VIC1       5935.1        -15.9751     5961.77002
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                   con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

class nempy.historical_inputs.mms_db.**InputsByIntervalDateTime**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs by INTERVAL_DATETIME.

Methods:

get_data(date_time)	Retrieves data for the specified date_time e.g.
add_data(year, month)	"Download data for the given table and time, appends to any existing data.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.

get_data(*date_time*)

Retrieves data for the specified date_time e.g. 2019/01/01 11:55:00”

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = InputsByIntervalDateTime(table_name='EXAMPLE', table_columns=[
↪'INTERVAL_DATETIME', 'INITIALMW'],
...                                   table_primary_keys=['INTERVAL_DATETIME'],␣
↪con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the add_data method to add historical data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...     'INTERVAL_DATETIME': ['2019/01/01 11:55:00', '2019/01/01 12:00:00'],
...     'INITIALMW': [1.0, 2.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

When we call get_data the output is filtered by INTERVAL_DATETIME.

```
>>> print(table.get_data(date_time='2019/01/01 12:00:00'))
     INTERVAL_DATETIME  INITIALMW
0  2019/01/01 12:00:00        2.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters
`date_time` (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

`add_data`(*year*, *month*)
"Download data for the given table and time, appends to any existing data.

Note: This method and its documentation is inherited from the _MultiDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MultiDataSource(table_name='DISPATCHREGIONSUM',
...     table_columns=['SETTLEMENTDATE', 'REGIONID', 'TOTALDEMAND',
...                    'DEMANDFORECAST', 'INITIALSUPPLY'],
...     table_primary_keys=['SETTLEMENTDATE', 'REGIONID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.add_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DISPATCHREGIONSUM order by SETTLEMENTDATE DESC limit␣
↪1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
       SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/02/01 00:00:00     VIC1       5935.1        -15.9751     5961.77002
```

If we subsequently add data from an earlier month the old data remains in the table, in addition to the new data.

```
>>> table.add_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/02/01 00:00:00     VIC1       5935.1        -15.9751     5961.77002
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                        con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

class nempy.historical_inputs.mms_db.**InputsByDay**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs by SETTLEMENTDATE, where inputs are stored on a daily basis.

Methods:

get_data(date_time)	Retrieves data for the specified date_time e.g.
add_data(year, month)	"Download data for the given table and time, appends to any existing data.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.

get_data(*date_time*)

Retrieves data for the specified date_time e.g. 2019/01/01 11:55:00, where inputs are stored on daily basis.

Note that a market day begins with the first 5 min interval as 04:05:00, there for if and input date_time of 2019/01/01 04:05:00 is given inputs where the SETTLEMENDATE is 2019/01/01 00:00:00 will be retrieved and if a date_time of 2019/01/01 04:00:00 or earlier is given then inputs where the SETTLEMENDATE is 2018/12/31 00:00:00 will be retrieved.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = InputsByDay(table_name='EXAMPLE', table_columns=['SETTLEMENTDATE',
↪'INITIALMW'],
...                     table_primary_keys=['SETTLEMENTDATE'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the add_data method to add historical data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...     'SETTLEMENTDATE': ['2019/01/01 00:00:00', '2019/01/02 00:00:00'],
...     'INITIALMW': [1.0, 2.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

When we call get_data the output is filtered by SETTLEMENTDATE and the results from the appropriate market day starting at 04:05:00 are retrieved. In the results below note when the output changes

```
>>> print(table.get_data(date_time='2019/01/01 12:00:00'))
        SETTLEMENTDATE  INITIALMW
0  2019/01/01 00:00:00        1.0
```

```
>>> print(table.get_data(date_time='2019/01/02 04:00:00'))
        SETTLEMENTDATE  INITIALMW
0  2019/01/01 00:00:00        1.0
```

```
>>> print(table.get_data(date_time='2019/01/02 04:05:00'))
        SETTLEMENTDATE  INITIALMW
0  2019/01/02 00:00:00        2.0
```

```
>>> print(table.get_data(date_time='2019/01/02 12:00:00'))
        SETTLEMENTDATE  INITIALMW
0  2019/01/02 00:00:00        2.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters
date_time (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

add_data(*year*, *month*)

"Download data for the given table and time, appends to any existing data.

Note: This method and its documentation is inherited from the _MultiDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MultiDataSource(table_name='DISPATCHREGIONSUM',
...   table_columns=['SETTLEMENTDATE', 'REGIONID', 'TOTALDEMAND',
...                  'DEMANDFORECAST', 'INITIALSUPPLY'],
...   table_primary_keys=['SETTLEMENTDATE', 'REGIONID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.add_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DISPATCHREGIONSUM order by SETTLEMENTDATE DESC limit␣
↪1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
       SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/02/01 00:00:00     VIC1       5935.1        -15.9751     5961.77002
```

If we subsequently add data from an earlier month the old data remains in the table, in addition to the new data.

```
>>> table.add_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       SETTLEMENTDATE REGIONID  TOTALDEMAND  DEMANDFORECAST  INITIALSUPPLY
0  2020/02/01 00:00:00     VIC1       5935.1        -15.9751     5961.77002
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                   con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

class nempy.historical_inputs.mms_db.**InputsStartAndEnd**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs by START_DATE and END_DATE.

Methods:

get_data(date_time)	Retrieves data for the specified date_time by START_DATE and END_DATE.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.
set_data(year, month)	"Download data for the given table and time, replace any existing data.

get_data(*date_time*)

Retrieves data for the specified date_time by START_DATE and END_DATE.

Records with a START_DATE before or equal to the date_times and an END_DATE after the date_time will be returned.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = InputsStartAndEnd(table_name='EXAMPLE', table_columns=['START_DATE',
↪ 'END_DATE', 'INITIALMW'],
...                             table_primary_keys=['START_DATE'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the add_data method to add historical data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...     'START_DATE': ['2019/01/01 00:00:00', '2019/01/02 00:00:00'],
...     'END_DATE': ['2019/01/02 00:00:00', '2019/01/03 00:00:00'],
...     'INITIALMW': [1.0, 2.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

When we call get_data the output is filtered by START_DATE and END_DATE.

```
>>> print(table.get_data(date_time='2019/01/01 00:00:00'))
            START_DATE             END_DATE  INITIALMW
0  2019/01/01 00:00:00  2019/01/02 00:00:00        1.0
```

```
>>> print(table.get_data(date_time='2019/01/01 12:00:00'))
            START_DATE             END_DATE  INITIALMW
0  2019/01/01 00:00:00  2019/01/02 00:00:00        1.0
```

```
>>> print(table.get_data(date_time='2019/01/02 00:00:00'))
            START_DATE             END_DATE  INITIALMW
0  2019/01/02 00:00:00  2019/01/03 00:00:00        2.0
```

```
>>> print(table.get_data(date_time='2019/01/02 00:12:00'))
            START_DATE             END_DATE  INITIALMW
0  2019/01/02 00:00:00  2019/01/03 00:00:00        2.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters
date_time (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                   con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

set_data(*year*, *month*)

"Download data for the given table and time, replace any existing data.

Note: This method and its documentation is inherited from the _SingleDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _SingleDataSource(table_name='DUDETAILSUMMARY',
...                           table_columns=['DUID', 'START_DATE',
↪'CONNECTIONPOINTID', 'REGIONID'],
...                           table_primary_keys=['START_DATE', 'DUID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.set_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DUDETAILSUMMARY order by START_DATE DESC limit 1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
      DUID           START_DATE CONNECTIONPOINTID REGIONID
0  URANQ11  2020/02/04 00:00:00            NURQ1U     NSW1
```

However if we subsequently set data from a previous date then any existing data will be replaced. Note the change in the most recent record in the data set below.

```
>>> table.set_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       DUID           START_DATE CONNECTIONPOINTID REGIONID
0  WEMENSF1  2019/03/04 00:00:00            VWES2W     VIC1
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.

- **month** (*int*) – The month to download data for.

Return type
None

```
class nempy.historical_inputs.mms_db.InputsByMatchDispatchConstraints(table_name, table_columns, table_primary_keys, con)
```

Manages retrieving dispatch inputs by matching against the DISPATCHCONSTRAINTS table

Methods:

get_data(date_time)	Retrieves data for the specified date_time by matching against the DISPATCHCONSTRAINT table.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.
set_data(year, month)	"Download data for the given table and time, replace any existing data.

get_data(*date_time*)

Retrieves data for the specified date_time by matching against the DISPATCHCONSTRAINT table.

First the DISPATCHCONSTRAINT table is filtered by SETTLEMENTDATE and then the contents of the classes table is matched against that.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = InputsByMatchDispatchConstraints(table_name='EXAMPLE',
...                             table_columns=['GENCONID', 'EFFECTIVEDATE',
↪'VERSIONNO', 'RHS'],
...                             table_primary_keys=['GENCONID', 'EFFECTIVEDATE',
↪'VERSIONNO'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the set_data method to add historical data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...   'GENCONID': ['X', 'X', 'Y', 'Y'],
...   'EFFECTIVEDATE': ['2019/01/02 00:00:00', '2019/01/03 00:00:00', '2019/01/
```

(continues on next page)

(continued from previous page)

```
↪01 00:00:00',
...                         '2019/01/03 00:00:00'],
...     'VERSIONNO': [1, 2, 2, 3],
...     'RHS': [1.0, 2.0, 2.0, 3.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

```
>>> data = pd.DataFrame({
...     'SETTLEMENTDATE' : ['2019/01/02 00:00:00', '2019/01/02 00:00:00', '2019/
↪01/03 00:00:00',
...                           '2019/01/03 00:00:00'],
...     'CONSTRAINTID': ['X', 'Y', 'X', 'Y'],
...     'GENCONID_EFFECTIVEDATE': ['2019/01/02 00:00:00', '2019/01/01 00:00:00',
↪'2019/01/03 00:00:00',
...                                   '2019/01/03 00:00:00'],
...     'GENCONID_VERSIONNO': [1, 2, 2, 3]})
```

```
>>> _ = data.to_sql('DISPATCHCONSTRAINT', con=con, if_exists='append',␣
↪index=False)
```

When we call get_data the output is filtered by the contents of DISPATCHCONSTRAINT.

```
>>> print(table.get_data(date_time='2019/01/02 00:00:00'))
  GENCONID        EFFECTIVEDATE VERSIONNO  RHS
0        X  2019/01/02 00:00:00         1  1.0
1        Y  2019/01/01 00:00:00         2  2.0
```

```
>>> print(table.get_data(date_time='2019/01/03 00:00:00'))
  GENCONID        EFFECTIVEDATE VERSIONNO  RHS
0        X  2019/01/03 00:00:00         2  2.0
1        Y  2019/01/03 00:00:00         3  3.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters
date_time (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                    con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

set_data(*year*, *month*)

"Download data for the given table and time, replace any existing data.

Note: This method and its documentation is inherited from the _SingleDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _SingleDataSource(table_name='DUDETAILSUMMARY',
...                           table_columns=['DUID', 'START_DATE',
↪'CONNECTIONPOINTID', 'REGIONID'],
...                           table_primary_keys=['START_DATE', 'DUID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.set_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DUDETAILSUMMARY order by START_DATE DESC limit 1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
      DUID           START_DATE CONNECTIONPOINTID REGIONID
0  URANQ11  2020/02/04 00:00:00            NURQ1U     NSW1
```

However if we subsequently set data from a previous date then any existing data will be replaced. Note the change in the most recent record in the data set below.

```
>>> table.set_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       DUID           START_DATE CONNECTIONPOINTID REGIONID
0  WEMENSF1  2019/03/04 00:00:00            VWES2W     VIC1
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

class nempy.historical_inputs.mms_db.**InputsByEffectiveDateVersionNoAndDispatchInterconnector**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs by EFFECTTIVEDATE and VERSIONNO.

Methods:

get_data(date_time)	Retrieves data for the specified date_time by EFFECTTIVEDATE and VERSIONNO.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.
set_data(year, month)	"Download data for the given table and time, replace any existing data.

get_data(*date_time*)

Retrieves data for the specified date_time by EFFECTTIVEDATE and VERSIONNO.

For each unique record (by the remaining primary keys, not including EFFECTTIVEDATE and VERSIONNO) the record with the most recent EFFECTIVEDATE

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical_inputs.db')
```

Create the table object.

```
>>> table = InputsByEffectiveDateVersionNoAndDispatchInterconnector(table_name=
↪'EXAMPLE',
...                               table_columns=['INTERCONNECTORID', 'EFFECTIVEDATE
↪', 'VERSIONNO', 'INITIALMW'],
...                               table_primary_keys=['INTERCONNECTORID',
↪'EFFECTIVEDATE', 'VERSIONNO'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the set_data method to add historical_inputs data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...     'INTERCONNECTORID': ['X', 'X', 'Y', 'Y'],
...     'EFFECTIVEDATE': ['2019/01/02 00:00:00', '2019/01/03 00:00:00', '2019/01/
↪01 00:00:00',
...                       '2019/01/03 00:00:00'],
...     'VERSIONNO': [1, 2, 2, 3],
...     'INITIALMW': [1.0, 2.0, 2.0, 3.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

We also need to add data to DISPATCHINTERCONNECTORRES because the results of the get_data method are filtered against this table

```
>>> data = pd.DataFrame({
...     'INTERCONNECTORID': ['X', 'X', 'Y'],
...     'SETTLEMENTDATE': ['2019/01/02 00:00:00', '2019/01/03 00:00:00', '2019/01/
↪02 00:00:00']})
```

```
>>> _ = data.to_sql('DISPATCHINTERCONNECTORRES', con=con, if_exists='append',␣
↪index=False)
```

When we call get_data the output is filtered by the contents of DISPATCHCONSTRAINT.

```
>>> print(table.get_data(date_time='2019/01/02 00:00:00'))
  INTERCONNECTORID         EFFECTIVEDATE VERSIONNO  INITIALMW
0                X  2019/01/02 00:00:00         1        1.0
1                Y  2019/01/01 00:00:00         2        2.0
```

In the next interval interconnector Y is not present in DISPATCHINTERCONNECTORRES.

```
>>> print(table.get_data(date_time='2019/01/03 00:00:00'))
  INTERCONNECTORID         EFFECTIVEDATE VERSIONNO  INITIALMW
0                X  2019/01/03 00:00:00         2        2.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical_inputs.db')
```

Parameters
date_time (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

create_table_in_sqlite_db()
Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                   con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

set_data(*year*, *month*)

"Download data for the given table and time, replace any existing data.

Note: This method and its documentation is inherited from the _SingleDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _SingleDataSource(table_name='DUDETAILSUMMARY',
...                          table_columns=['DUID', 'START_DATE',
↪'CONNECTIONPOINTID', 'REGIONID'],
...                          table_primary_keys=['START_DATE', 'DUID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.set_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DUDETAILSUMMARY order by START_DATE DESC limit 1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
      DUID           START_DATE CONNECTIONPOINTID REGIONID
0  URANQ11  2020/02/04 00:00:00            NURQ1U     NSW1
```

However if we subsequently set data from a previous date then any existing data will be replaced. Note the change in the most recent record in the data set below.

```
>>> table.set_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       DUID           START_DATE CONNECTIONPOINTID REGIONID
0  WEMENSF1  2019/03/04 00:00:00            VWES2W     VIC1
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

class nempy.historical_inputs.mms_db.**InputsByEffectiveDateVersionNo**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs by EFFECTTIVEDATE and VERSIONNO.

Methods:

get_data(date_time)	Retrieves data for the specified date_time by EFFECTTIVEDATE and VERSIONNO.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.
set_data(year, month)	"Download data for the given table and time, replace any existing data.

get_data(*date_time*)

Retrieves data for the specified date_time by EFFECTTIVEDATE and VERSIONNO.

For each unique record (by the remaining primary keys, not including EFFECTTIVEDATE and VERSIONNO) the record with the most recent EFFECTIVEDATE

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = InputsByEffectiveDateVersionNo(table_name='EXAMPLE',
...                               table_columns=['DUID', 'EFFECTIVEDATE', 'VERSIONNO
↪', 'INITIALMW'],
...                               table_primary_keys=['DUID', 'EFFECTIVEDATE',
↪'VERSIONNO'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the set_data method to add historical data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...   'DUID': ['X', 'X', 'Y', 'Y'],
...   'EFFECTIVEDATE': ['2019/01/02 00:00:00', '2019/01/03 00:00:00', '2019/01/
↪01 00:00:00',
...                     '2019/01/03 00:00:00'],
...   'VERSIONNO': [1, 2, 2, 3],
...   'INITIALMW': [1.0, 2.0, 2.0, 3.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

When we call get_data the output is filtered by most recent effective date and highest version no.

```
>>> print(table.get_data(date_time='2019/01/02 00:00:00'))
  DUID        EFFECTIVEDATE VERSIONNO  INITIALMW
0    X  2019/01/02 00:00:00         1        1.0
1    Y  2019/01/01 00:00:00         2        2.0
```

In the next interval interconnector Y is not present in DISPATCHINTERCONNECTORRES.

```
>>> print(table.get_data(date_time='2019/01/03 00:00:00'))
  DUID        EFFECTIVEDATE VERSIONNO  INITIALMW
0    X  2019/01/03 00:00:00         2        2.0
1    Y  2019/01/03 00:00:00         3        3.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters
date_time (*str*) – Should be of format '%Y/%m/%d %H:%M:%S', and always a round 5 min interval e.g. 2019/01/01 11:55:00.

Return type
pd.DataFrame

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                   con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

set_data(*year*, *month*)

"Download data for the given table and time, replace any existing data.

Note: This method and its documentation is inherited from the _SingleDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _SingleDataSource(table_name='DUDETAILSUMMARY',
...                           table_columns=['DUID', 'START_DATE',
↪'CONNECTIONPOINTID', 'REGIONID'],
...                           table_primary_keys=['START_DATE', 'DUID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.set_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DUDETAILSUMMARY order by START_DATE DESC limit 1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
      DUID           START_DATE CONNECTIONPOINTID REGIONID
0  URANQ11  2020/02/04 00:00:00            NURQ1U     NSW1
```

However if we subsequently set data from a previous date then any existing data will be replaced. Note the change in the most recent record in the data set below.

```
>>> table.set_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       DUID           START_DATE CONNECTIONPOINTID REGIONID
0  WEMENSF1  2019/03/04 00:00:00            VWES2W     VIC1
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

class nempy.historical_inputs.mms_db.**InputsNoFilter**(*table_name*, *table_columns*, *table_primary_keys*, *con*)

Manages retrieving dispatch inputs where no filter is require.

Methods:

get_data()	Retrieves all data in the table.
create_table_in_sqlite_db()	Creates a table in the sqlite database that the object has a connection to.
set_data(year, month)	"Download data for the given table and time, replace any existing data.

get_data()

Retrieves all data in the table.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical_inputs.db')
```

Create the table object.

```
>>> table = InputsNoFilter(table_name='EXAMPLE', table_columns=['DUID',
↪'INITIALMW'],
...                          table_primary_keys=['DUID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Normally you would use the set_data method to add historical_inputs data, but here we will add data directly to the database so some simple example data can be added.

```
>>> data = pd.DataFrame({
...   'DUID': ['X', 'Y'],
...   'INITIALMW': [1.0, 2.0]})
```

```
>>> _ = data.to_sql('EXAMPLE', con=con, if_exists='append', index=False)
```

When we call get_data all data in the table is returned.

```
>>> print(table.get_data())
  DUID  INITIALMW
0    X        1.0
1    Y        2.0
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical_inputs.db')
```

Return type
pd.DataFrame

create_table_in_sqlite_db()

Creates a table in the sqlite database that the object has a connection to.

Note: This method and its documentation is inherited from the _MMSTable class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _MMSTable(table_name='EXAMPLE', table_columns=['DUID', 'BIDTYPE'],␣
↪table_primary_keys=['DUID'],
...                   con=con)
```

Create the corresponding table in the sqlite database, note this step many not be needed if you have connected to an existing database.

```
>>> table.create_table_in_sqlite_db()
```

Now a table exists in the database, but its empty.

```
>>> print(pd.read_sql("Select * from example", con=con))
Empty DataFrame
Columns: [DUID, BIDTYPE]
Index: []
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

set_data(*year*, *month*)

"Download data for the given table and time, replace any existing data.

Note: This method and its documentation is inherited from the _SingleDataSource class.

Examples

```
>>> import sqlite3
>>> import os
```

Set up a database or connect to an existing one.

```
>>> con = sqlite3.connect('historical.db')
```

Create the table object.

```
>>> table = _SingleDataSource(table_name='DUDETAILSUMMARY',
...                           table_columns=['DUID', 'START_DATE',
↪'CONNECTIONPOINTID', 'REGIONID'],
...                           table_primary_keys=['START_DATE', 'DUID'], con=con)
```

Create the table in the database.

```
>>> table.create_table_in_sqlite_db()
```

Downloading data from http://nemweb.com.au/#mms-data-model into the table.

```
>>> table.set_data(year=2020, month=1)
```

Now the database should contain data for this table that is up to date as the end of Janurary.

```
>>> query = "Select * from DUDETAILSUMMARY order by START_DATE DESC limit 1;"
```

```
>>> print(pd.read_sql_query(query, con=con))
      DUID           START_DATE CONNECTIONPOINTID REGIONID
0  URANQ11  2020/02/04 00:00:00            NURQ1U     NSW1
```

However if we subsequently set data from a previous date then any existing data will be replaced. Note the change in the most recent record in the data set below.

```
>>> table.set_data(year=2019, month=1)
```

```
>>> print(pd.read_sql_query(query, con=con))
       DUID           START_DATE CONNECTIONPOINTID REGIONID
0  WEMENSF1  2019/03/04 00:00:00            VWES2W     VIC1
```

Clean up by closing the database and deleting if its no longer needed.

```
>>> con.close()
>>> os.remove('historical.db')
```

Parameters

- **year** (*int*) – The year to download data for.
- **month** (*int*) – The month to download data for.

Return type
None

5.3 loaders

Classes:

RawInputsLoader(nemde_xml_cache_manager, ...)	Provides single interface for accessing raw historical inputs.

class nempy.historical_inputs.loaders.**RawInputsLoader**(*nemde_xml_cache_manager*, *market_management_system_database*)

Provides single interface for accessing raw historical inputs.

Examples

```
>>> import sqlite3
```

```
>>> from nempy.historical_inputs import mms_db
>>> from nempy.historical_inputs import xml_cache
```

For the RawInputsLoader to work we need to construct a database and inputs cache for it to load inputs from and then pass the interfaces to these to the inputs loader.

```
>>> con = sqlite3.connect('market_management_system.db')
>>> mms_db_manager = mms_db.DBManager(connection=con)
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
```

In this example the database and cache have already been populated so the input loader can be created straight away.

```
>>> inputs_loader = RawInputsLoader(xml_cache_manager, mms_db_manager)
```

Then we set the dispatch interval that we want to load inputs from.

```
>>> inputs_loader.set_interval('2019/01/01 00:00:00')
```

And then we can load some inputs.

```
>>> inputs_loader.get_unit_volume_bids()
         DUID    BIDTYPE  MAXAVAIL  ENABLEMENTMIN  ENABLEMENTMAX  LOWBREAKPOINT ␣
↪HIGHBREAKPOINT  BANDAVAIL1  BANDAVAIL2  BANDAVAIL3  BANDAVAIL4  BANDAVAIL5 ␣
↪BANDAVAIL6  BANDAVAIL7  BANDAVAIL8  BANDAVAIL9  BANDAVAIL10  RAMPDOWNRATE ␣
↪RAMPUPRATE
0      AGLHAL     ENERGY     173.0            0.0            0.0            0.0  ␣
↪          0.0         0.0         0.0         0.0         0.0         0.0  ␣
↪0.0        60.0         0.0         0.0        160.0         720.0       720.0
1      AGLSOM     ENERGY     160.0            0.0            0.0            0.0  ␣
↪          0.0         0.0         0.0         0.0         0.0         0.0  ␣
↪0.0         0.0         0.0         0.0        170.0         480.0       480.0
2     ANGAST1     ENERGY      43.0            0.0            0.0            0.0  ␣
↪          0.0         0.0         0.0         0.0         0.0         0.0 ␣
↪50.0         0.0         0.0         0.0         50.0         840.0       840.0
3       APD01   LOWER5MIN       0.0            0.0            0.0            0.0  ␣
↪          0.0         0.0         0.0         0.0         0.0         0.0  ␣
↪0.0         0.0         0.0         0.0        300.0           0.0         0.0
4       APD01  LOWER60SEC       0.0            0.0            0.0            0.0  ␣
↪          0.0         0.0         0.0         0.0         0.0         0.0  ␣
↪0.0         0.0         0.0         0.0        300.0           0.0         0.0
...       ...         ...       ...            ...            ...            ...  ␣
↪          ...         ...         ...         ...         ...         ...         .
↪..         ...         ...         ...          ...           ...         ...
1021    YWPS4  LOWER6SEC      25.0          250.0          385.0          275.0  ␣
↪        385.0        15.0        10.0         0.0         0.0         0.0  ␣
↪0.0         0.0         0.0         0.0          0.0           0.0         0.0
1022    YWPS4  RAISE5MIN       0.0          250.0          390.0          250.0  ␣
↪        380.0         0.0         0.0         0.0         0.0         5.0  ␣
```

(continues on next page)

(continued from previous page)

```
↪0.0          0.0          5.0          0.0          10.0          0.0          0.0
1023     YWPS4     RAISEREG       15.0          250.0          385.0          250.0  ␣
↪         370.0          0.0          0.0          0.0          0.0          0.0  ␣
↪0.0          5.0         10.0          0.0           5.0          0.0          0.0
1024     YWPS4   RAISE60SEC       10.0          220.0          400.0          220.0  ␣
↪         390.0          0.0          0.0          0.0          0.0          0.0  ␣
↪5.0          5.0          0.0          0.0          10.0          0.0          0.0
1025     YWPS4    RAISE6SEC       15.0          220.0          405.0          220.0  ␣
↪         390.0          0.0          0.0          0.0         10.0          5.0  ␣
↪0.0          0.0          0.0          0.0          10.0          0.0          0.0

[1026 rows x 19 columns]
```

Methods:

set_interval(interval)	Set the interval to load inputs for.
get_unit_initial_conditions()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_unit_initial_conditions*
get_unit_volume_bids()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_unit_volume_bids*
get_unit_price_bids()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.BIDDAYOFFER_D.get_data*
get_unit_details()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.DUDETAILSUMMARY.get_data*
get_agc_enablement_limits()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.DISPATCHLOAD.get_data*
get_UIGF_values()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_UIGF_values*
get_violations()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_violations*
get_constraint_violation_prices()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_violation_prices*
get_constraint_rhs()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_rhs*
get_constraint_type()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_type*
get_constraint_region_lhs()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_region_lhs*
get_constraint_unit_lhs()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_unit_lhs*
get_constraint_interconnector_lhs()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_interconnector_lhs*
get_market_interconnectors()	Direct interface to nempy.historical_inputs.mms_db.DBManager.MNSP_INTERCONNECTOR.get_data
get_market_interconnector_link_bid_availability()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_market_interconnector_link_bid_availability*
get_interconnector_constraint_parameters()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.INTERCONNECTORCONSTRAINT.get_data*
get_interconnector_definitions()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.INTERCONNECTOR.get_data*
get_regional_loads()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.DISPATCHREGIONSUM.get_data*
get_interconnector_loss_segments()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.LOSSMODEL.get_data*
get_interconnector_loss_parameters()	Direct interface to *nempy.historical_inputs.mms_db.DBManager.LOSSFACTORMODEL.get_data*
get_unit_fast_start_parameters()	Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.*

set_interval(*interval*)

Set the interval to load inputs for.

Examples

For an example see the *class level documentation*

Parameters
interval (*str*) – In the format '%Y/%m/%d %H:%M:%S'

get_unit_initial_conditions()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_unit_initial_conditions*

get_unit_volume_bids()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_unit_volume_bids*

get_unit_price_bids()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.BIDDAYOFFER_D.get_data*

get_unit_details()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.DUDETAILSUMMARY.get_data*

get_agc_enablement_limits()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.DISPATCHLOAD.get_data*

get_UIGF_values()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_UIGF_values*

get_violations()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_violations*

get_constraint_violation_prices()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_violation_prices*

get_constraint_rhs()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_rhs*

get_constraint_type()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_type*

get_constraint_region_lhs()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_region_lhs*

get_constraint_unit_lhs()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_unit_lhs*

get_constraint_interconnector_lhs()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_constraint_interconnector_lhs*

get_market_interconnectors()

Direct interface to nempy.historical_inputs.mms_db.DBManager.MNSP_INTERCONNECTOR.get_data

get_market_interconnector_link_bid_availability()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_market_interconnector_link_bid_availability*

get_interconnector_constraint_parameters()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.INTERCONNECTORCONSTRAINT.get_data*

get_interconnector_definitions()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.INTERCONNECTOR.get_data*

get_regional_loads()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.DISPATCHREGIONSUM.get_data*

get_interconnector_loss_segments()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.LOSSMODEL.get_data*

get_interconnector_loss_parameters()

Direct interface to *nempy.historical_inputs.mms_db.DBManager.LOSSFACTORMODEL.get_data*

get_unit_fast_start_parameters()

Direct interface to *nempy.historical_inputs.xml_cache.XMLCacheManager.get_unit_fast_start_parameters*

is_over_constrained_dispatch_rerun()

Checks if the over constrained dispatch rerun process was used by AEMO to dispatch this interval.

Examples

```
>>> import sqlite3
```

```
>>> from nempy.historical_inputs import mms_db
>>> from nempy.historical_inputs import xml_cache
```

For the RawInputsLoader to work we need to construct a database and inputs cache for it to load inputs from and then pass the interfaces to these to the inputs loader.

```
>>> con = sqlite3.connect('market_management_system.db')
>>> mms_db_manager = mms_db.DBManager(connection=con)
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
```

In this example the database and cache have already been populated so the input loader can be created straight away.

```
>>> inputs_loader = RawInputsLoader(xml_cache_manager, mms_db_manager)
```

Then we set the dispatch interval that we want to load inputs from.

```
>>> inputs_loader.set_interval('2019/01/01 00:00:00')
```

And then we can load some inputs.

```
>>> inputs_loader.is_over_constrained_dispatch_rerun()
False
```

Return type
bool

5.4 units

Exceptions:

MethodCallOrderError	Raise for calling methods in incompatible order.

Classes:

UnitData(raw_input_loader)	Loads unit related raw inputs and preprocess them for compatibility with *nempy.markets.SpotMarket*

exception nempy.historical_inputs.units.**MethodCallOrderError**

Raise for calling methods in incompatible order.

class nempy.historical_inputs.units.**UnitData**(*raw_input_loader*)

Loads unit related raw inputs and preprocess them for compatibility with *nempy.markets.SpotMarket*

Examples

This example shows the setup used for the examples in the class methods.

```
>>> import sqlite3
>>> from nempy.historical_inputs import mms_db
>>> from nempy.historical_inputs import xml_cache
>>> from nempy.historical_inputs import loaders
```

The UnitData class requries a RawInputsLoader instance.

```
>>> con = sqlite3.connect('market_management_system.db')
>>> mms_db_manager = mms_db.DBManager(connection=con)
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> inputs_loader = loaders.RawInputsLoader(xml_cache_manager, mms_db_manager)
>>> inputs_loader.set_interval('2019/01/10 12:05:00')
```

Create the UnitData instance.

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_unit_bid_availability()
         unit  capacity
0      AGLHAL     170.0
1      AGLSOM     160.0
```

(continues on next page)

(continued from previous page)

```
2       ANGAST1      44.0
23       BALBG1       0.0
33       BALBL1       0.0
...         ...       ...
989    YARWUN_1     165.0
990       YWPS1     380.0
999       YWPS2     180.0
1008      YWPS3     350.0
1017      YWPS4     340.0

[218 rows x 2 columns]
```

Methods:

get_unit_bid_availability()	Get the bid in maximum availability for scheduled units.
get_unit_uigf_limits()	Get the maximum availability predicted by the unconstrained intermittent generation forecast.
get_ramp_rates_used_for_energy_dispatch([...])	Get ramp rates used for constraining energy dispatch.
get_as_bid_ramp_rates()	Get ramp rates used as bid by units.
get_initial_unit_output()	Get unit outputs at the start of the dispatch interval.
get_fast_start_profiles_for_dispatch([...])	Get the parameters needed to construct the fast dispatch inflexibility profiles used for dispatch.
get_unit_info()	Get unit information.
get_processed_bids()	Get processed unit bids.
add_fcas_trapezium_constraints()	Load the fcas trapezium constraints into the UnitData class so subsequent method calls can access them.
get_fcas_max_availability()	Get the unit bid maximum availability of each service.
get_fcas_regulation_trapeziums()	Get the unit bid FCAS trapeziums for regulation services.
get_scada_ramp_down_rates_of_lower_reg_units([...])	Get the scada ramp down rates for unit with a lower regulation bid.
get_scada_ramp_up_rates_of_raise_reg_units([...])	Get the scada ramp up rates for unit with a raise regulation bid.
get_contingency_services()	Get the unit bid FCAS trapeziums for contingency services.

get_unit_bid_availability()

Get the bid in maximum availability for scheduled units.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_unit_bid_availability()
        unit  capacity
```

(continues on next page)

(continued from previous page)

```
0       AGLHAL     170.0
1       AGLSOM     160.0
2      ANGAST1      44.0
23      BALBG1       0.0
33      BALBL1       0.0
...        ...       ...
989   YARWUN_1     165.0
990      YWPS1     380.0
999      YWPS2     180.0
1008     YWPS3     350.0
1017     YWPS4     340.0

[218 rows x 2 columns]
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
capacity	unit bid in max availability, in MW, (as *str*)

Return type
pd.DataFrame

get_unit_uigf_limits()

Get the maximum availability predicted by the unconstrained intermittent generation forecast.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_unit_uigf_limits()
        unit  capacity
0      ARWF1    18.654
1    BALDHWF1   11.675
2      BANN1    53.661
3     BLUFF1     8.655
4     BNGSF1    98.877
..       ...       ...
57     WGWF1     7.649
58   WHITSF1     6.075
59   WOODLWN1   11.659
60     WRSF1    20.000
61     WRWF1     7.180

[62 rows x 2 columns]
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
capacity	the forecast max availability, in MW, (as *str*)

Return type
pd.DataFrame

get_ramp_rates_used_for_energy_dispatch(*run_type='no_fast_start_units'*)

Get ramp rates used for constraining energy dispatch.

The minimum of bid in ramp rates and scada telemetered ramp rates are used. If 'no_fast_start_units' is given as the run_type then no extra process is applied to the ramp rates based on the fast start inflexibility profiles. If 'fast_start_first_run' is given then the ramp rates of units starting in fast start modes 0, 1, and 2 are excluded. If 'fast_start_second_run' is given then the ramp rates of units ending the interval in fast start modes 0, 1, and 2 are excluded, and the ramp rates of units that started interval in mode 2 or smaller, but end in mode 3 or greater, have there ramp rates adjusted to account for speeding a portion of the interval constrained from ramping up by their dispatch inflexibility profile.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_ramp_rates_used_for_energy_dispatch()
          unit  initial_output  ramp_up_rate  ramp_down_rate
0       AGLHAL        0.000000    720.000000      720.000000
1       AGLSOM        0.000000    480.000000      480.000000
2      ANGAST1        0.000000    840.000000      840.000000
3        ARWF1       15.800001   1200.000000      600.000000
4       BALBG1        0.000000   6000.000000     6000.000000
..         ...             ...           ...             ...
275   YARWUN_1      157.019989      0.000000        0.000000
276      YWPS1      383.959503    177.750006      177.750006
277      YWPS2      180.445572    177.750006      177.750006
278      YWPS3      353.460754    175.499997      175.499997
279      YWPS4      338.782288    180.000000      180.000000

[280 rows x 4 columns]
```

Parameters
run_type (*str specifying the run type should be one of 'no_fast_start_units', 'fast_start_first_run', or*) – 'fast_start_second_run'.

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Return type
pd.DataFrame

get_as_bid_ramp_rates()

Get ramp rates used as bid by units.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_as_bid_ramp_rates()
          unit  ramp_up_rate  ramp_down_rate
0       AGLHAL         720.0           720.0
1       AGLSOM         480.0           480.0
2      ANGAST1         840.0           840.0
9        ARWF1        1200.0           600.0
23      BALBG1        6000.0          6000.0
...        ...           ...             ...
989   YARWUN_1           0.0             0.0
990     YWPS1          180.0           180.0
999     YWPS2          180.0           180.0
1008    YWPS3          180.0           180.0
1017    YWPS4          180.0           180.0

[280 rows x 3 columns]
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Return type
pd.DataFrame

get_initial_unit_output()

Get unit outputs at the start of the dispatch interval.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_initial_unit_output()
         unit  initial_output
0      AGLHAL        0.000000
1      AGLSOM        0.000000
2     ANGAST1        0.000000
3       APD01        0.000000
4       ARWF1       15.800001
..        ...             ...
283  YARWUN_1      157.019989
284     YWPS1      383.959503
285     YWPS2      180.445572
286     YWPS3      353.460754
287     YWPS4      338.782288

[288 rows x 2 columns]
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)

Return type
pd.DataFrame

get_fast_start_profiles_for_dispatch(*unconstrained_dispatch=None*)

Get the parameters needed to construct the fast dispatch inflexibility profiles used for dispatch.

If the results of an non fast start constrained dispatch run are provided then these are used to commit fast start units starting the interval in mode zero, when the they have a non-zero dispatch result.

For more info on fast start dispatch inflexibility profiles `see AEMO docs`.

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
end_mode	the fast start mode the unit will end the dispatch interval in, (as *np.int64*)
time_in_end_mode	the amount of time the unit will have spend in the end mode at the end of the dispatch interval, (as *np.float64*)
mode_two_length	the length the units mode two, in minutes (as *np.float64*)
mode_four_length	the length the units mode four, in minutes (as *np.float64*)
min_loading	the mininum opperating level of the unit during mode three, in MW, (as *no.float64*)

Return type
pd.DataFrame

`get_unit_info()`

Get unit information.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> unit_data.get_unit_info()
          unit region dispatch_type  loss_factor
0       AGLHAL    SA1     generator     0.971500
1      AGLNOW1   NSW1     generator     1.003700
2     AGLSITA1   NSW1     generator     1.002400
3       AGLSOM   VIC1     generator     0.984743
4      ANGAST1    SA1     generator     1.005674
..         ...    ...           ...          ...
477      YWNL1   VIC1     generator     0.957300
478      YWPS1   VIC1     generator     0.969600
479      YWPS2   VIC1     generator     0.957300
480      YWPS3   VIC1     generator     0.957300
481      YWPS4   VIC1     generator     0.957300

[482 rows x 4 columns]
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
region	the market region in which the unit is located, (as *str*)
dis-patch_type	whether the unit is a 'generator' or 'load', (as *str*)
loss_factor	the combined unit transmission and distribution loss_factor, (as np.float64)

Return type
pd.DataFrame

get_processed_bids()

Get processed unit bids.

The bids are processed by scaling for AGC enablement limits, scaling for scada ramp rates, scaling for the unconstrained intermittent generation forecast and enforcing the preconditions for enabling FCAS bids. For more info on these processes `see AEMO docs`.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = UnitData(inputs_loader)
```

```
>>> volume_bids, price_bids = unit_data.get_processed_bids()
```

```
>>> volume_bids
         unit    service    1      2    3     4    5     6     7     8    9    ␣
↪10
0      AGLHAL     energy  0.0    0.0  0.0   0.0  0.0   0.0  60.0   0.0  0.0  160.
↪0
1      AGLSOM     energy  0.0    0.0  0.0   0.0  0.0   0.0   0.0   0.0  0.0  170.
↪0
2     ANGAST1     energy  0.0    0.0  0.0   0.0  0.0  50.0   0.0   0.0  0.0   50.
↪0
9       ARWF1     energy  0.0  241.0  0.0   0.0  0.0   0.0   0.0   0.0  0.0    0.
↪0
23     BALBG1     energy  0.0    0.0  0.0   0.0  0.0   0.0   0.0   0.0  0.0   30.
↪0
..        ...        ...  ...    ...  ...   ...  ...   ...   ...   ...  ...   ..
↪.
364     YWPS4   raise_6s  0.0    0.0  0.0  10.0  5.0   0.0   0.0   0.0  0.0   10.
↪0
365     YWPS4  lower_reg  0.0    0.0  0.0   0.0  0.0   0.0   0.0  20.0  0.0    0.
↪0
366     YWPS4  raise_reg  0.0    0.0  0.0   0.0  0.0   0.0   5.0  10.0  0.0    5.
↪0
369    SWAN_E  lower_reg  0.0    0.0  0.0   0.0  0.0   0.0   5.0   0.0  0.0   52.
↪0
```

(continues on next page)

(continued from previous page)

```
370   SWAN_E  raise_reg  0.0    0.0  0.0   5.0  0.0   0.0   3.0   0.0  0.0   49.
↪0

[591 rows x 12 columns]
```

```
>>> price_bids
        unit     service           1          2           3           4        ␣
↪  5           6           7           8           9            10
0     AGLHAL      energy  -971.50000   0.000000  270.863915  358.298915  406.
↪873915  484.593915  562.313915  1326.641540  10277.372205  13600.018785
1     AGLSOM      energy  -984.74292   0.000000   83.703148  108.321721  142.
↪787723  279.666989  444.119057   985.727663  13097.937562  14278.732950
2    ANGAST1      energy -1005.67390   0.000000  125.709237  201.335915  300.
↪887574  382.135969  593.337544  1382.650761  10678.245470  14582.271550
3      ARWF1      energy  -969.10000 -63.001191    1.996346    4.002383    8.
↪004766   15.999841   31.999682    63.999364    127.998728  14051.950000
4     BALBG1      energy  -994.80000   0.000000   19.915896   47.372376   75.
↪177036  109.447896  298.440000   443.133660  10047.489948  14424.600000
..       ...         ...         ...        ...         ...         ...        ␣
↪...         ...         ...          ...           ...           ...
586  ASQENC1    raise_6s     0.03000   0.300000    0.730000    0.990000    1.
↪980000    5.000000    9.900000    17.700000    100.000000  10000.000000
587  ASTHYD1    raise_6s     0.00000   0.490000    1.450000    4.950000    9.
↪950000   15.000000   60.000000   200.000000   1000.000000  14000.000000
588   VENUS1  raise_5min     0.00000   1.000000    2.780000    3.980000    4.
↪980000    8.600000    9.300000    14.600000     20.000000   1000.000000
589   VENUS1   raise_60s     0.00000   1.000000    2.780000    3.980000    4.
↪980000    8.600000    9.300000    14.600000     20.000000   1000.000000
590   VENUS1    raise_6s     0.01000   0.600000    2.780000    3.980000    4.
↪980000    8.600000    9.300000    14.000000     20.000000   1000.000000

[591 rows x 12 columns]
```

Multiple Returns

volume_bids : pd.DataFrame

Columns:	Description:
unit	unique identifier for units, (as *str*)
service	the service the bid applies to, (as *str*)
1	the volume bid the first bid band, in MW, (as *np.float64*)
:	
10	the volume in the tenth bid band, in MW, (as *np.float64*)

price_bids : pd.DataFrame

Columns:	Description:
unit	unique identifier for units, (as *str*)
service	the service the bid applies to, (as *str*)
1	the price of the first bid band, in MW, (as *np.float64*)
:	
10	the price of the the tenth bid band, in MW, (as *np.float64*)

add_fcas_trapezium_constraints()

Load the fcas trapezium constraints into the UnitData class so subsequent method calls can access them.

Examples

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
```

If we try and call add_fcas_trapezium_constraints before calling get_processed_bids we get an error.

```
>>> unit_data.add_fcas_trapezium_constraints()
Traceback (most recent call last):
   ...
nempy.historical_inputs.units.MethodCallOrderError: Call get_processed_bids␣
↪before add_fcas_trapezium_constraints.
```

After calling get_processed_bids it goes away.

```
>>> volume_bids, price_bids =  unit_data.get_processed_bids()
```

```
>>> unit_data.add_fcas_trapezium_constraints()
```

If we try and access the trapezium constraints before calling this method we get an error.

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
>>> unit_data.get_fcas_max_availability()
Traceback (most recent call last):
   ...
nempy.historical_inputs.units.MethodCallOrderError: Call add_fcas_trapezium_
↪constraints before get_fcas_max_availability.
```

After calling it the error goes away.

```
>>> volume_bids, price_bids = unit_data.get_processed_bids()
>>> unit_data.add_fcas_trapezium_constraints()
```

```
>>> unit_data.get_fcas_max_availability()
        unit     service  max_availability
0      APD01  raise_5min              34.0
1      APD01   raise_60s              34.0
2      APD01    raise_6s              17.0
```

(continues on next page)

(continued from previous page)

```
3    ASNENC1  raise_5min             12.0
4    ASNENC1   raise_60s              4.0
..       ...         ...              ...
364    YWPS4    raise_6s             15.0
365    YWPS4   lower_reg             15.0
366    YWPS4   raise_reg             15.0
369   SWAN_E   lower_reg             10.0
370   SWAN_E   raise_reg             25.0

[311 rows x 3 columns]
```

Return type
None

Raises
MethodCallOrderError – if called before get_processed_bids

get_fcas_max_availability()

Get the unit bid maximum availability of each service.

Examples

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
```

Required calls before calling get_fcas_max_availability.

```
>>> volume_bids, price_bids =  unit_data.get_processed_bids()
>>> unit_data.add_fcas_trapezium_constraints()
```

Now facs max availibility can be accessed.

```
>>> unit_data.get_fcas_max_availability()
        unit     service  max_availability
0      APD01  raise_5min             34.0
1      APD01   raise_60s             34.0
2      APD01    raise_6s             17.0
3    ASNENC1  raise_5min             12.0
4    ASNENC1   raise_60s              4.0
..       ...         ...              ...
364    YWPS4    raise_6s             15.0
365    YWPS4   lower_reg             15.0
366    YWPS4   raise_reg             15.0
369   SWAN_E   lower_reg             10.0
370   SWAN_E   raise_reg             25.0

[311 rows x 3 columns]
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
service	the service the bid applies to, (as *str*)
max_availability	the unit bid maximum availability, in MW, (as *np.float64*)

Return type
pd.DataFrame

Raises
MethodCallOrderError – if the method is called before add_fcas_trapezium_constraints.

get_fcas_regulation_trapeziums()

Get the unit bid FCAS trapeziums for regulation services.

Examples

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
```

Required calls before calling get_fcas_regulation_trapeziums.

```
>>> volume_bids, price_bids =  unit_data.get_processed_bids()
>>> unit_data.add_fcas_trapezium_constraints()
```

Now facs max availibility can be accessed.

```
>>> unit_data.get_fcas_regulation_trapeziums()
         unit    service  max_availability  enablement_min  low_break_point ␣
↪high_break_point  enablement_max
16       BW01  lower_reg         35.015640       309.27185       344.287490    ␣
↪   520.80701       520.80701
17       BW01  raise_reg         35.015640       309.27185       309.271850    ␣
↪   485.79137       520.80701
24   CALL_B_1  lower_reg         15.000000       180.00000       195.000000    ␣
↪   270.30002       270.30002
25   CALL_B_1  raise_reg         15.000000       180.00000       180.000000    ␣
↪   205.00000       220.00000
55       ER01  lower_reg         24.906273       490.02502       514.931293    ␣
↪   680.00000       680.00000
..        ...        ...               ...             ...              ...    ␣
↪         ...             ...
359     YWPS3  raise_reg         14.625000       250.00000       250.000000    ␣
↪   370.37500       385.00000
365     YWPS4  lower_reg         15.000000       250.00000       265.000000    ␣
↪   385.00000       385.00000
366     YWPS4  raise_reg         15.000000       250.00000       250.000000    ␣
↪   370.00000       385.00000
369    SWAN_E  lower_reg         10.000000       145.00000       202.000000    ␣
↪   362.50000       362.50000
370    SWAN_E  raise_reg         25.000000       145.00000       145.000000    ␣
↪   305.50000       362.50000
```

(continues on next page)

(continued from previous page)

```
[75 rows x 7 columns]
```

Returns

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
service	the regulation service being offered, (as *str*)
max_availability	the maximum volume of the contingency service, in MW, (as *np.float64*)
enablement_min	the energy dispatch level at which the unit can begin to provide the regulation service, in MW, (as *np.float64*)
low_break_point	the energy dispatch level at which the unit can provide the full regulation service offered, in MW, (as *np.float64*)
high_break_point	the energy dispatch level at which the unit can no longer provide the full regulation service offered, in MW, (as *np.float64*)
enablement_max	the energy dispatch level at which the unit can no longer provide any regulation service, in MW, (as *np.float64*)

Return type
pd.DataFrame

Raises
MethodCallOrderError – if the method is called before add_fcas_trapezium_constraints.

get_scada_ramp_down_rates_of_lower_reg_units(*run_type='no_fast_start_units'*)

Get the scada ramp down rates for unit with a lower regulation bid.

Only units with scada ramp rates and a lower regulation bid that passes enablement criteria are returned.

Examples

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
```

Required calls before calling get_scada_ramp_down_rates_of_lower_reg_units.

```
>>> volume_bids, price_bids =  unit_data.get_processed_bids()
>>> unit_data.add_fcas_trapezium_constraints()
```

Now the method can be called.

```
>>> unit_data.get_scada_ramp_down_rates_of_lower_reg_units().head()
       unit  initial_output  ramp_down_rate
36      BW01      425.125000      420.187683
40  CALL_B_1      219.699997      240.000000
74      ER01      636.000000      298.875275
76      ER03      678.925049      297.187500
77      ER04      518.550049      298.312225
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Return type
pd.DataFrame

Raises
MethodCallOrderError – if the method is called before add_fcas_trapezium_constraints.

get_scada_ramp_up_rates_of_raise_reg_units(*run_type='no_fast_start_units'*)

Get the scada ramp up rates for unit with a raise regulation bid.

Only units with scada ramp rates and a raise regulation bid that passes enablement criteria are returned.

Examples

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
```

Required calls before calling get_scada_ramp_up_rates_of_raise_reg_units.

```
>>> volume_bids, price_bids =  unit_data.get_processed_bids()
>>> unit_data.add_fcas_trapezium_constraints()
```

Now the method can be called.

```
>>> unit_data.get_scada_ramp_up_rates_of_raise_reg_units().head()
       unit  initial_output  ramp_up_rate
36      BW01      425.125000    420.187683
40  CALL_B_1      219.699997    240.000000
74      ER01      636.000000    299.999542
76      ER03      678.925049    297.750092
77      ER04      518.550049    298.875275
```

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)

Return type
pd.DataFrame

Raises
MethodCallOrderError – if the method is called before add_fcas_trapezium_constraints.

get_contingency_services()
Get the unit bid FCAS trapeziums for contingency services.

Examples

```
>>> inputs_loader = _test_setup()
>>> unit_data = UnitData(inputs_loader)
```

Required calls before calling get_contingency_services.

```
>>> volume_bids, price_bids =  unit_data.get_processed_bids()
>>> unit_data.add_fcas_trapezium_constraints()
```

Now facs max availibility can be accessed.

```
>>> unit_data.get_contingency_services()
        unit     service  max_availability  enablement_min  low_break_point ␣
↪high_break_point  enablement_max
0      APD01  raise_5min              34.0             0.0              0.0     ␣
↪         0.0             0.0
1      APD01   raise_60s              34.0             0.0              0.0     ␣
↪         0.0             0.0
2      APD01    raise_6s              17.0             0.0              0.0     ␣
↪         0.0             0.0
3    ASNENC1  raise_5min              12.0             0.0              0.0     ␣
↪         0.0             0.0
4    ASNENC1   raise_60s               4.0             0.0              0.0     ␣
↪         0.0             0.0
..       ...         ...               ...             ...              ...     ␣
↪         ...             ...
360    YWPS4  lower_5min              15.0           250.0            265.0     ␣
↪       385.0           385.0
361    YWPS4   lower_60s              20.0           250.0            270.0     ␣
↪       385.0           385.0
362    YWPS4    lower_6s              25.0           250.0            275.0     ␣
↪       385.0           385.0
363    YWPS4   raise_60s              10.0           220.0            220.0     ␣
↪       390.0           400.0
```

(continues on next page)

(continued from previous page)

```
364     YWPS4     raise_6s              15.0           220.0            220.0     ␣
↪       390.0            405.0

[236 rows x 7 columns]
```

Returns

Columns:	Description:
unit	unique identifier of a dispatch unit, (as *str*)
service	the contingency service being offered, (as *str*)
max_availability	the maximum volume of the contingency service, in MW, (as *np.float64*)
enablement_min	the energy dispatch level at which the unit can begin to provide the regulation service, in MW, (as *np.float64*)
low_break_point	the energy dispatch level at which the unit can provide the full regulation service offered, in MW, (as *np.float64*)
high_break_point	the energy dispatch level at which the unit can no longer provide the full regulation service offered, in MW, (as *np.float64*)
enablement_max	the energy dispatch level at which the unit can no longer provide any regulation service, in MW, (as *np.float64*)

Return type
pd.DataFrame

Raises
`MethodCallOrderError` – if the method is called before add_fcas_trapezium_constraints.

5.5 interconnectors

Classes:

`InterconnectorData`(raw_input_loader)	Loads interconnector related raw inputs and preprocess them for compatibility with `nempy.markets.SpotMarket`

Functions:

`create_loss_functions`(...)	Creates a loss function for each interconnector.

class nempy.historical_inputs.interconnectors.**InterconnectorData**(*raw_input_loader*)

Loads interconnector related raw inputs and preprocess them for compatibility with *nempy.markets.SpotMarket*

Examples

This example shows the setup used for the examples in the class methods. This setup is used to create a RawInputsLoader by calling the function _test_setup.

```
>>> import sqlite3
>>> from nempy.historical_inputs import mms_db
>>> from nempy.historical_inputs import xml_cache
>>> from nempy.historical_inputs import loaders
```

The InterconnectorData class requries a RawInputsLoader instance.

```
>>> con = sqlite3.connect('market_management_system.db')
>>> mms_db_manager = mms_db.DBManager(connection=con)
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> inputs_loader = loaders.RawInputsLoader(xml_cache_manager, mms_db_manager)
>>> inputs_loader.set_interval('2019/01/10 12:05:00')
```

Create a InterconnectorData instance.

```
>>> interconnector_data = InterconnectorData(inputs_loader)
```

```
>>> interconnector_data.get_interconnector_definitions()
  interconnector from_region to_region     min     max       link  from_region_loss_
↪factor  to_region_loss_factor  generic_constraint_factor
0           V-SA        VIC1       SA1  -850.0   950.0       V-SA                 ␣
↪1.0000              1.0000                          1
1      N-Q-MNSP1        NSW1      QLD1  -264.0   264.0  N-Q-MNSP1                 ␣
↪1.0000              1.0000                          1
2      NSW1-QLD1        NSW1      QLD1 -1659.0  1229.0  NSW1-QLD1                 ␣
↪1.0000              1.0000                          1
3      V-S-MNSP1        VIC1       SA1  -270.0   270.0  V-S-MNSP1                 ␣
↪1.0000              1.0000                          1
5      VIC1-NSW1        VIC1      NSW1 -2299.0  2399.0  VIC1-NSW1                 ␣
↪1.0000              1.0000                          1
0      T-V-MNSP1        TAS1      VIC1     0.0   478.0    BLNKTAS                 ␣
↪1.0000              0.9839                          1
1      T-V-MNSP1        VIC1      TAS1     0.0   478.0    BLNKVIC                 ␣
↪0.9839              1.0000                         -1
```

Parameters
inputs_manager (*historical_spot_market_inputs.DBManager*) –

Methods:

get_interconnector_loss_model()	Returns inputs in the format needed to set interconnector losses in the SpotMarket class.
get_interconnector_definitions()	Returns inputs in the format needed to create interconnectors in the SpotMarket class.

get_interconnector_loss_model()

Returns inputs in the format needed to set interconnector losses in the SpotMarket class.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> interconnector_data = InterconnectorData(inputs_loader)
```

```
>>> loss_function, interpolation_break_points =              interconnector_
↪data.get_interconnector_loss_model()
```

```
>>> print(loss_function)
  interconnector       link                                      loss_function ␣
↪from_region_loss_share
0           V-SA       V-SA  <function InterconnectorData.get_interconnecto... ␣
↪                      0.78
1       N-Q-MNSP1  N-Q-MNSP1  <function InterconnectorData.get_interconnecto... ␣
↪                      0.66
2       NSW1-QLD1  NSW1-QLD1  <function InterconnectorData.get_interconnecto... ␣
↪                      0.68
3       V-S-MNSP1  V-S-MNSP1  <function InterconnectorData.get_interconnecto... ␣
↪                      0.67
4       VIC1-NSW1  VIC1-NSW1  <function InterconnectorData.get_interconnecto... ␣
↪                      0.32
5       T-V-MNSP1    BLNKTAS  <function InterconnectorData.get_interconnecto... ␣
↪                      1.00
6       T-V-MNSP1    BLNKVIC  <function InterconnectorData.get_interconnecto... ␣
↪                      1.00
```

```
>>> print(interpolation_break_points)
    interconnector     link  loss_segment  break_point
0             V-SA     V-SA             1       -851.0
1             V-SA     V-SA             2       -835.0
2             V-SA     V-SA             3       -820.0
3             V-SA     V-SA             4       -805.0
4             V-SA     V-SA             5       -790.0
..             ...      ...           ...          ...
599      T-V-MNSP1  BLNKVIC           -80       -546.0
600      T-V-MNSP1  BLNKVIC           -81       -559.0
601      T-V-MNSP1  BLNKVIC           -82       -571.0
602      T-V-MNSP1  BLNKVIC           -83       -583.0
603      T-V-MNSP1  BLNKVIC           -84       -595.0

[604 rows x 4 columns]
```

Multiple Returns

loss_functions : pd.DataFrame

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
from_region_loss_share	The fraction of loss occuring in the from region, 0.0 to 1.0, (as *np.float64*)
loss_function	A function that takes a flow, in MW as a float and returns the losses in MW, (as *callable*)

interpolation_break_points : pd.DataFrame

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
loss_segment	unique identifier of a loss segment on an interconnector basis, (as *np.float64*)
break_point	points between which the loss function will be linearly interpolated, in MW (as *np.float64*)

get_interconnector_definitions()

Returns inputs in the format needed to create interconnectors in the SpotMarket class.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> interconnector_data = InterconnectorData(inputs_loader)
```

```
>>> interconnector_data.get_interconnector_definitions()
  interconnector from_region to_region     min     max       link  from_region_
↪loss_factor  to_region_loss_factor  generic_constraint_factor
0         V-SA        VIC1       SA1  -850.0   950.0       V-SA                 ␣
↪    1.0000                 1.0000                          1
1      N-Q-MNSP1        NSW1      QLD1  -264.0   264.0  N-Q-MNSP1                 ␣
↪    1.0000                 1.0000                          1
2      NSW1-QLD1        NSW1      QLD1 -1659.0  1229.0  NSW1-QLD1                 ␣
↪    1.0000                 1.0000                          1
3      V-S-MNSP1        VIC1       SA1  -270.0   270.0  V-S-MNSP1                 ␣
↪    1.0000                 1.0000                          1
5      VIC1-NSW1        VIC1      NSW1 -2299.0  2399.0  VIC1-NSW1                 ␣
↪    1.0000                 1.0000                          1
0      T-V-MNSP1        TAS1      VIC1     0.0   478.0    BLNKTAS                 ␣
```

(continues on next page)

(continued from previous page)

```
↪    1.0000              0.9839                    1
1      T-V-MNSP1        VIC1       TAS1      0.0    478.0     BLNKVIC          ␣
↪    0.9839              1.0000                   -1
```

Returns

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
to_region	the region that receives power when flow is in the positive direction, (as *str*)
from_region	the region that power is drawn from when flow is in the positive direction, (as *str*)
max	the maximum power flow on the interconnector, in MW (as *np.float64*)
min	the minimum power flow on the interconnector, if power can flow neative direction then this will be negative, in MW (as *np.float64*)
from_region_loss_factor	the loss factor between the from end of the interconnector and the regional reference node, (as *np.float*)
to_region_loss_factor	the loss factor between the to end of the interconnector and the regional reference node, (as *np.float*)

Return type
pd.DataFrame

`nempy.historical_inputs.interconnectors.`**`create_loss_functions`**(*interconnector_coefficients*, *demand_coefficients*, *demand*)

Creates a loss function for each interconnector.

Transforms the dynamic demand dependendent interconnector loss functions into functions that only depend on interconnector flow. i.e takes the function f and creates g by pre-calculating the demand dependent terms.

> f(inter_flow, flow_coefficient, nsw_demand, nsw_coefficient, qld_demand, qld_coefficient) = inter_losses

becomes

> g(inter_flow) = inter_losses

The mathematics of the demand dependent loss functions is described in the `Marginal Loss Factors documentation section 3 to 5`.

Examples

```
>>> import pandas as pd
```

Some arbitrary regional demands.

```
>>> demand = pd.DataFrame({
...   'region': ['VIC1', 'NSW1', 'QLD1', 'SA1'],
...   'loss_function_demand': [6000.0 , 7000.0, 5000.0, 3000.0]})
```

Loss model details from 2020 Jan NEM web LOSSFACTORMODEL file

```
>>> demand_coefficients = pd.DataFrame({
...   'interconnector': ['NSW1-QLD1', 'NSW1-QLD1', 'VIC1-NSW1',
...                      'VIC1-NSW1', 'VIC1-NSW1'],
...   'region': ['NSW1', 'QLD1', 'NSW1', 'VIC1', 'SA1'],
...   'demand_coefficient': [-0.00000035146, 0.000010044,
...                           0.000021734, -0.000031523,
...                          -0.000065967]})
```

Loss model details from 2020 Jan NEM web INTERCONNECTORCONSTRAINT file

```
>>> interconnector_coefficients = pd.DataFrame({
...   'interconnector': ['NSW1-QLD1', 'VIC1-NSW1'],
...   'loss_constant': [0.9529, 1.0657],
...   'flow_coefficient': [0.00019617, 0.00017027],
...   'from_region_loss_share': [0.5, 0.5]})
```

Create the loss functions

```
>>> loss_functions = create_loss_functions(interconnector_coefficients,
...                                        demand_coefficients, demand)
```

Lets use one of the loss functions, first get the loss function of VIC1-NSW1 and call it g

```
>>> g = loss_functions[loss_functions['interconnector'] == 'VIC1-NSW1']['loss_
↪function'].iloc[0]
```

Calculate the losses at 600 MW flow

```
>>> print(g(600.0))
-70.87199999999996
```

Now for NSW1-QLD1

```
>>> h = loss_functions[loss_functions['interconnector'] == 'NSW1-QLD1']['loss_
↪function'].iloc[0]
```

```
>>> print(h(600.0))
35.70646799999993
```

Parameters

- **interconnector_coefficients** (*pd.DataFrame*) –

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
loss_constant	the constant term in the interconnector loss factor equation, (as *np.float64*)
flow_coefficient	the coefficient of the interconnector flow variable in the loss factor equation (as *np.float64*)
from_region_loss_share	the proportion of loss attribute to the from region, remainer are attributed to the to region, (as *np.float64*)

- **demand_coefficients** (*pd.DataFrame*) –

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
region	the market region whose demand the coefficient applies too (as *str*)
demand_coefficient	the coefficient of regional demand variable in the loss factor equation, (as *np.float64*)

- **demand** (*pd.DataFrame*) –

Columns:	Description:
region	unique identifier of a region, (as *str*)
loss_function_demand	the estimated regional demand, as calculated by initial supply + demand forecast, in MW (as *np.float64*)

Returns

loss_functions

Columns:	Description:
interconnector	unique identifier of a interconnector, (as *str*)
loss_function	a *function* object that takes interconnector flow (as *float*) an input and returns interconnector losses (as *float*).

Return type

pd.DataFrame

5.6 demand

Classes:

DemandData(raw_inputs_loader)	Loads demand related raw data and preprocess it for complatibility with the SpotMarket class.

class nempy.historical_inputs.demand.**DemandData**(*raw_inputs_loader*)

Loads demand related raw data and preprocess it for complatibility with the SpotMarket class.

Examples

The DemandData class requries a RawInputsLoader instance.

```
>>> import sqlite3
>>> from nempy.historical_inputs import mms_db
>>> from nempy.historical_inputs import xml_cache
>>> from nempy.historical_inputs import loaders
>>> con = sqlite3.connect('market_management_system.db')
>>> mms_db_manager = mms_db.DBManager(connection=con)
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> inputs_loader = loaders.RawInputsLoader(xml_cache_manager, mms_db_manager)
>>> inputs_loader.set_interval('2019/01/10 12:05:00')
```

```
>>> demand_data = DemandData(inputs_loader)
```

```
>>> demand_data.get_operational_demand()
  region   demand
0   NSW1  8540.33
1   QLD1  7089.69
2    SA1  1019.21
3   TAS1  1070.89
4   VIC1  4500.71
```

Parameters
raw_inputs_loader –

Methods:

get_operational_demand()	Get the operational demand used to determine the regional energy dispatch constraints.

get_operational_demand()

Get the operational demand used to determine the regional energy dispatch constraints.

Examples

See class level example.

Returns

Columns:	Description:
region	unique identifier of a market region, (as *str*)
demand	the non dispatchable demand the region, in MW, (as *np.float64*)
loss_function_demand	the measure of demand used when creating interconnector loss functions, in MW, (as *np.float64*)

Return type
pd.DataFrame

5.7 constraints

Classes:

ConstraintData(raw_inputs_loader)	Loads generic constraint related raw inputs and preprocess them for compatibility with *nempy.markets.SpotMarket*

class nempy.historical_inputs.constraints.**ConstraintData**(*raw_inputs_loader*)

Loads generic constraint related raw inputs and preprocess them for compatibility with *nempy.markets.SpotMarket*

Examples

This example shows the setup used for the examples in the class methods. This setup is used to create a RawInputsLoader by calling the function _test_setup.

```
>>> import sqlite3
>>> from nempy.historical_inputs import mms_db
>>> from nempy.historical_inputs import xml_cache
>>> from nempy.historical_inputs import loaders
```

The InterconnectorData class requries a RawInputsLoader instance.

```
>>> con = sqlite3.connect('market_management_system.db')
>>> mms_db_manager = mms_db.DBManager(connection=con)
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> inputs_loader = loaders.RawInputsLoader(xml_cache_manager, mms_db_manager)
>>> inputs_loader.set_interval('2019/01/01 00:00:00')
```

Create a InterconnectorData instance.

```
>>> constraint_data = ConstraintData(inputs_loader)
```

```
>>> constraint_data.get_rhs_and_type_excluding_regional_fcas_constraints()
                      set           rhs type
0                #BANN1_E     32.000000   <=
1               #BNGSF2_E      3.000000   <=
2             #CROWLWF1_E     43.000000   <=
3              #CSPVPS1_E     29.000000   <=
4              #DAYDSF1_E      0.000000   <=
..                    ...           ...  ...
704           V_OWF_NRB_0  10000.001000   <=
705   V_OWF_TGTSNRBHTN_30  10030.000000   <=
706          V_S_NIL_ROCOF    812.280029   <=
707            V_T_NIL_BL1    478.000000   <=
708          V_T_NIL_FCSPS    425.154024   <=

[574 rows x 3 columns]
```

Parameters
inputs_manager (*historical_spot_market_inputs.DBManager*) –

Methods:

get_rhs_and_type_excluding_regional_fcas_constraints()	Get the rhs values and types for generic constraints, excludes regional FCAS constraints.
get_rhs_and_type()	Get the rhs values and types for generic constraints.
get_unit_lhs()	Get the lhs coefficients of units.
get_interconnector_lhs()	Get the lhs coefficients of interconnectors.
get_region_lhs()	Get the lhs coefficients of regions.
get_fcas_requirements()	Get constraint details needed for setting FCAS requirements.
get_violation_costs()	Get the violation costs for generic constraints.
get_constraint_violation_prices()	Get the violation costs of non-generic constraint groups.
is_over_constrained_dispatch_rerun()	Get a boolean indicating if the over constrained dispatch rerun process was used for this interval.

get_rhs_and_type_excluding_regional_fcas_constraints()

Get the rhs values and types for generic constraints, excludes regional FCAS constraints.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_rhs_and_type_excluding_regional_fcas_constraints()
                      set           rhs type
0                #BANN1_E     32.000000   <=
```

(continues on next page)

(continued from previous page)

```
1                #BNGSF2_E      3.000000   <=
2              #CROWLWF1_E     43.000000   <=
3               #CSPVPS1_E     29.000000   <=
4               #DAYDSF1_E      0.000000   <=
..                     ...           ...  ...
704            V_OWF_NRB_0  10000.001000   <=
705   V_OWF_TGTSNRBHTN_30  10030.000000   <=
706          V_S_NIL_ROCOF    812.280029   <=
707            V_T_NIL_BL1    478.000000   <=
708          V_T_NIL_FCSPS    425.154024   <=

[574 rows x 3 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the constraint set, (as *str*)
type	the direction of the constraint >=, <= or =, (as *str*)
rhs	the right hand side value of the constraint, (as *np.float64*)

Return type
pd.DataFrame

get_rhs_and_type()
Get the rhs values and types for generic constraints.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_rhs_and_type()
                       set           rhs type
0               #BANN1_E     32.000000   <=
1                #BNGSF2_E      3.000000   <=
2              #CROWLWF1_E     43.000000   <=
3               #CSPVPS1_E     29.000000   <=
4               #DAYDSF1_E      0.000000   <=
..                     ...           ...  ...
704            V_OWF_NRB_0  10000.001000   <=
705   V_OWF_TGTSNRBHTN_30  10030.000000   <=
706          V_S_NIL_ROCOF    812.280029   <=
707            V_T_NIL_BL1    478.000000   <=
708          V_T_NIL_FCSPS    425.154024   <=

[709 rows x 3 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the constraint set, (as *str*)
type	the direction of the constraint >=, <= or =, (as *str*)
rhs	the right hand side value of the constraint, (as *np.float64*)

Return type
pd.DataFrame

get_unit_lhs()
Get the lhs coefficients of units.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_unit_lhs()
                      set      unit service  coefficient
0               #BANN1_E     BANN1  energy          1.0
1              #BNGSF2_E    BNGSF2  energy          1.0
2             #CROWLWF1_E  CROWLWF1  energy          1.0
3              #CSPVPS1_E   CSPVPS1  energy          1.0
4              #DAYDSF1_E   DAYDSF1  energy          1.0
...                   ...       ...     ...          ...
5864       V_ARWF_FSTTRP_5     ARWF1  energy          1.0
5865       V_MTGBRAND_33WT  MTGELWF1  energy          1.0
5866      V_OAKHILL_TFB_42  OAKLAND1  energy          1.0
5867           V_OWF_NRB_0  OAKLAND1  energy          1.0
5868  V_OWF_TGTSNRBHTN_30  OAKLAND1  energy          1.0

[5869 rows x 4 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
unit	the unit whose variables will be mapped to the lhs, (as *str*)
service	the service whose variables will be mapped to the lhs, (as *str*)
coefficient	the lhs coefficient (as *np.float64*)

Return type
pd.DataFrame

get_interconnector_lhs()

Get the lhs coefficients of interconnectors.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_interconnector_lhs()
                     set interconnector  coefficient
0               DATASNAP       N-Q-MNSP1          1.0
1        DATASNAP_DFS_LS       N-Q-MNSP1          1.0
2      DATASNAP_DFS_NCAN       N-Q-MNSP1          1.0
3    DATASNAP_DFS_NCWEST       N-Q-MNSP1          1.0
4      DATASNAP_DFS_NNTH       N-Q-MNSP1          1.0
..                   ...             ...          ...
631      V^^S_NIL_TBSE_1            V-SA          1.0
632      V^^S_NIL_TBSE_2            V-SA          1.0
633        V_S_NIL_ROCOF            V-SA          1.0
634          V_T_NIL_BL1       T-V-MNSP1         -1.0
635        V_T_NIL_FCSPS       T-V-MNSP1         -1.0

[636 rows x 3 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
intercon-netor	the interconnetor whose variables will be mapped to the lhs, (as *str*)
coefficient	the lhs coefficient (as *np.float64*)

Return type
pd.DataFrame

get_region_lhs()

Get the lhs coefficients of regions.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_region_lhs()
                   set region    service  coefficient
```

(continues on next page)

(continued from previous page)

```
0        F_I+LREG_0120   NSW1  lower_reg          1.0
1        F_I+LREG_0120   QLD1  lower_reg          1.0
2        F_I+LREG_0120    SA1  lower_reg          1.0
3        F_I+LREG_0120   TAS1  lower_reg          1.0
4        F_I+LREG_0120   VIC1  lower_reg          1.0
..                 ...    ...        ...          ...
478   F_T+NIL_WF_TG_R5   TAS1  raise_reg          1.0
479   F_T+NIL_WF_TG_R6   TAS1   raise_6s          1.0
480  F_T+NIL_WF_TG_R60   TAS1  raise_60s          1.0
481      F_T+RREG_0050   TAS1  raise_reg          1.0
482    F_T_NIL_MINP_R6   TAS1   raise_6s          1.0

[483 rows x 4 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
region	the region whose variables will be mapped to the lhs, (as *str*)
service	the service whose variables will be mapped to the lhs, (as *str*)
coefficient	the lhs coefficient (as *np.float64*)

Return type
pd.DataFrame

get_fcas_requirements()

Get constraint details needed for setting FCAS requirements.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_fcas_requirements()
                   set    service region type      volume
0        F_I+LREG_0120  lower_reg   NSW1   >=  120.000000
1        F_I+LREG_0120  lower_reg   QLD1   >=  120.000000
2        F_I+LREG_0120  lower_reg    SA1   >=  120.000000
3        F_I+LREG_0120  lower_reg   TAS1   >=  120.000000
4        F_I+LREG_0120  lower_reg   VIC1   >=  120.000000
..                 ...        ...    ...  ...         ...
478   F_T+NIL_WF_TG_R5  raise_reg   TAS1   >=   62.899972
479   F_T+NIL_WF_TG_R6   raise_6s   TAS1   >=   67.073327
480  F_T+NIL_WF_TG_R60  raise_60s   TAS1   >=   83.841637
```

(continues on next page)

(continued from previous page)

```
481      F_T+RREG_0050  raise_reg   TAS1   >= -9950.000000
482     F_T_NIL_MINP_R6   raise_6s   TAS1   >=    35.000000

[483 rows x 5 columns]
```

Returns

Columns:	Description:
set	unique identifier of the requirement set, (as *str*)
service	the service or services the requirement set applies to (as *str*)
region	the regions that can contribute to meeting a requirement, (as *str*)
volume	the amount of service required, in MW, (as *np.float64*)
type	the direction of the constrain '=', '>=' or '<=', optional, a value of '=' is assumed if the column is missing (as *str*)

Return type
pd.DataFrame

get_violation_costs()

Get the violation costs for generic constraints.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_violation_costs()
                     set       cost
0              #BANN1_E  5220000.0
1             #BNGSF2_E  5220000.0
2            #CROWLWF1_E  5220000.0
3             #CSPVPS1_E  5220000.0
4             #DAYDSF1_E  5220000.0
..                   ...        ...
704           V_OWF_NRB_0  5220000.0
705  V_OWF_TGTSNRBHTN_30  5220000.0
706         V_S_NIL_ROCOF   507500.0
707           V_T_NIL_BL1  5220000.0
708         V_T_NIL_FCSPS   435000.0

[709 rows x 2 columns]
```

Returns

Columns:	Description:
set	the unique identifier of the constraint set to map the lhs coefficients to, (as *str*)
cost	the cost to the objective function of violating the constraint, (as *np.float64*)

Return type
pd.DataFrame

get_constraint_violation_prices()

Get the violation costs of non-generic constraint groups.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.get_constraint_violation_prices()
{'regional_demand': 2175000.0, 'interocnnector': 16675000.0, 'generic_constraint
↪': 435000.0, 'ramp_rate': 16747500.0, 'unit_capacity': 5365000.0, 'energy_
↪offer': 16457500.0, 'fcas_profile': 2247500.0, 'fcas_max_avail': 2247500.0,
↪'fcas_enablement_min': 1015000.0, 'fcas_enablement_max': 1015000.0, 'fast_
↪start': 16385000.0, 'mnsp_ramp_rate': 16747500.0, 'msnp_offer': 16457500.0,
↪'mnsp_capacity': 5292500.0, 'uigf': 5582500.0, 'voll': 14500.0, 'tiebreak':␣
↪1e-06}
```

Return type
dict

is_over_constrained_dispatch_rerun()

Get a boolean indicating if the over constrained dispatch rerun process was used for this interval.

Examples

```
>>> inputs_loader = _test_setup()
```

```
>>> unit_data = ConstraintData(inputs_loader)
```

```
>>> unit_data.is_over_constrained_dispatch_rerun()
False
```

Return type
bool

5.8 RHSCalc

Classes:

RHSCalc(xml_cache_manager)	Engine for calculating generic constraint right hand side (RHS) values from scratch based on the equations provided in the NEMDE xml input files.

class nempy.historical_inputs.rhs_calculator.**RHSCalc**(*xml_cache_manager*)

Engine for calculating generic constraint right hand side (RHS) values from scratch based on the equations provided in the NEMDE xml input files.

AEMO publishes the RHS values used in dispatch, however, those values are dynamically calculated by NEMDE and depend on inputs such as transmission line flows, generator on statuses, and generator output levels. This class allows the user to update the input values which the RHS equations depend on and then recalulate RHS values. The primary reason for implementing this functionality is to allow the Bass link switch run to be implemented using Nempy, which requires that the RHS values of a number of constraints to be recalculated for the case where the bass link frequency controller is not active.

The methodology for the calculation is based on the description in the Constraint Implementation Guidelines published by AEMO, `see AEMO doc`. The main limitation of the method implemented is that it does not allow for the calculation of constraints that use BRANCH operation. In 2013 there were three constraints using the branching operation (V^SML_NIL_3, V^SML_NSWRB_2, V^S_HYCP, Q^NIL_GC), and in 2023 it appears the branch operation is no longer in active use. While there are some difference between the RHS values produced, generally they are small,

Examples

```
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> xml_cache_manager.load_interval('2019/01/01 00:00:00')
>>> rhs_calculator = RHSCalc(xml_cache_manager)
```

Parameters
xml_cache_manager (*instance of nempy class XMLCacheManager*) –

Methods:

get_nemde_rhs(constraint_id)	Get the RHS values of a constraints as calculated by NEMDE.
compute_constraint_rhs(constraint_id)	Calculates the rhs values of the speficied constraint or list of constraints.
get_rhs_constraint_equations_that_depend_value(...)	A helper method used to find the which constraints' RHS depend on a given input value.
update_spd_id_value(spd_id, type, value)	Updates the value of one of the inputs which the RHS constraint equations depend on.

get_nemde_rhs(*constraint_id*)

Get the RHS values of a constraints as calculated by NEMDE. This method is implemented primarily to assist with testing.

Examples

```
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> xml_cache_manager.load_interval('2019/01/01 00:00:00')
>>> rhs_calculator = RHSCalc(xml_cache_manager)
>>> rhs_calculator.get_nemde_rhs("F_MAIN++NIL_BL_R60")
-10290.279635
```

Parameters
constraint_id (*str which is the unique ID of the constraint*) –

Return type
float

compute_constraint_rhs(*constraint_id*)
Calculates the rhs values of the speficied constraint or list of constraints.

Examples

```
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> xml_cache_manager.load_interval('2019/01/01 00:00:00')
>>> rhs_calculator = RHSCalc(xml_cache_manager)
>>> rhs_calculator.compute_constraint_rhs('F_MAIN++NIL_BL_R60')
-10290.737541856766
```

```
>>> rhs_calculator.compute_constraint_rhs(['F_MAIN++NIL_BL_R60', 'F_MAIN++NIL_
↪BL_R6'])
                  set           rhs
0  F_MAIN++NIL_BL_R60 -10290.737542
1   F_MAIN++NIL_BL_R6 -10581.475084
```

Parameters
constraint_id (*str or list[str] which is the unique ID of the constraint or a list of the strings which are*) – the constraint IDs

Return type
float or pandas DataFrame

get_rhs_constraint_equations_that_depend_value(*spd_id*, *type*)
A helper method used to find the which constraints' RHS depend on a given input value.

Examples

```
>>> xml_cache_manager = xml_cache.XMLCacheManager('nemde_cache_2014_12')
>>> xml_cache_manager.load_interval('2014/12/05 00:00:00')
>>> rhs_calculator = RHSCalc(xml_cache_manager)
>>> rhs_calculator.get_rhs_constraint_equations_that_depend_value('BL_FREQ_
↪ONSTATUS', 'W')
['F_MAIN++APD_TL_L5', 'F_MAIN++APD_TL_L6', 'F_MAIN++APD_TL_L60', 'F_MAIN++ML_L5_
↪0400', 'F_MAIN++ML_L5_APD', 'F_MAIN++ML_L60_0400', 'F_MAIN++ML_L60_APD', 'F_
```

(continues on next page)

(continued from previous page)

```
↪MAIN++ML_L6_0400', 'F_MAIN++ML_L6_APD', 'F_MAIN++NIL_DYN_LREG', 'F_MAIN++NIL_
↪DYN_RREG', 'F_MAIN++NIL_MG_R5', 'F_MAIN++NIL_MG_R6', 'F_MAIN++NIL_MG_R60', 'F_
↪MAIN+APD_TL_L5', 'F_MAIN+APD_TL_L6', 'F_MAIN+APD_TL_L60', 'F_MAIN+ML_L5_0400',
↪ 'F_MAIN+ML_L5_APD', 'F_MAIN+ML_L60_0400', 'F_MAIN+ML_L60_APD', 'F_MAIN+ML_L6_
↪0400', 'F_MAIN+ML_L6_APD', 'F_MAIN+NIL_DYN_LREG', 'F_MAIN+NIL_DYN_RREG', 'F_
↪MAIN+NIL_MG_R5', 'F_MAIN+NIL_MG_R6', 'F_MAIN+NIL_MG_R60', 'F_T++LREG_0050',
↪'F_T++NIL_BB_TG_R5', 'F_T++NIL_BB_TG_R6', 'F_T++NIL_BB_TG_R60', 'F_T++NIL_MG_
↪R5', 'F_T++NIL_MG_R6', 'F_T++NIL_MG_R60', 'F_T++NIL_ML_L5', 'F_T++NIL_ML_L6',
↪'F_T++NIL_ML_L60', 'F_T++NIL_TL_L5', 'F_T++NIL_TL_L6', 'F_T++NIL_TL_L60', 'F_
↪T++NIL_WF_TG_R5', 'F_T++NIL_WF_TG_R6', 'F_T++NIL_WF_TG_R60', 'F_T++RREG_0050',
↪ 'F_T+LREG_0050', 'F_T+NIL_BB_TG_R5', 'F_T+NIL_BB_TG_R6', 'F_T+NIL_BB_TG_R60',
↪ 'F_T+NIL_MG_R5', 'F_T+NIL_MG_R6', 'F_T+NIL_MG_R60', 'F_T+NIL_ML_L5', 'F_
↪T+NIL_ML_L6', 'F_T+NIL_ML_L60', 'F_T+NIL_TL_L5', 'F_T+NIL_TL_L6', 'F_T+NIL_TL_
↪L60', 'F_T+NIL_WF_TG_R5', 'F_T+NIL_WF_TG_R6', 'F_T+NIL_WF_TG_R60', 'F_T+RREG_
↪0050', 'T_V_NIL_BL1', 'V_T_NIL_BL1']
```

Parameters

- **spd_id** (*str, the ID of the value used in the NEMDE xml input file.*) –
- **type** (*str, the type of the value used in the NEMDE xml input file. See the Constraint Implementation Guidelines*) – published by AEMO for more information on SPD types, see AEMO doc

Return type

list[str] a list of strings detailing the constraits' whose RHS equations depend on the specified value.

update_spd_id_value(*spd_id*, *type*, *value*)

Updates the value of one of the inputs which the RHS constraint equations depend on.

Examples

```
>>> xml_cache_manager = xml_cache.XMLCacheManager('test_nemde_cache')
>>> xml_cache_manager.load_interval('2019/01/01 00:00:00')
>>> rhs_calculator = RHSCalc(xml_cache_manager)
>>> rhs_calculator.update_spd_id_value('220_GEN_INERTIA', 'A', '100.0')
```

Parameters

- **spd_id** (*str, the ID of the value used in the NEMDE xml input file.*) –
- **type** (*str, the type of the value used in the NEMDE xml input file. See the Constraint Implementation Guidelines*) – published by AEMO for more information on SPD types, see AEMO doc
- **value** (*str (detailing a float number) the new value to set the input to.*) –

CHAPTER

SIX

TIME_SEQUENTIAL MODULES

The module provides tools constructing time sequential models using nempy.

Functions:

construct_ramp_rate_parameters(...)	Combine dispatch and ramp rates into the ramp rate inputs compatible with the SpotMarket class.
create_seed_ramp_rate_parameters(...)	Combine historical dispatch and as bid ramp rates to get seed ramp rate parameters for a time sequential model.

nempy.time_sequential.**construct_ramp_rate_parameters**(*last_interval_dispatch*, *ramp_rates*)

Combine dispatch and ramp rates into the ramp rate inputs compatible with the SpotMarket class.

Examples

```
>>> last_interval_dispatch = pd.DataFrame({
... 'unit': ['A', 'A', 'B'],
... 'service': ['energy', 'raise_reg', 'energy'],
... 'dispatch': [45.0, 50.0, 88.0]})
```

```
>>> ramp_rates = pd.DataFrame({
... 'unit': ['A', 'B', 'C'],
... 'ramp_up_rate': [600.0, 1200.0, 700.0],
... 'ramp_down_rate': [600.0, 1200.0, 700.0]})
```

```
>>> construct_ramp_rate_parameters(last_interval_dispatch,
...                                ramp_rates)
  unit  initial_output  ramp_up_rate  ramp_down_rate
0    A            45.0         600.0           600.0
1    B            88.0        1200.0          1200.0
2    C             0.0         700.0           700.0
```

Parameters

- **last_interval_dispatch** (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier of a dispatch unit (as *str*)
service	the service being provided, optional, default 'energy', (as *str*)
dispatch	the dispatch target from the previous dispatch interval, in MW, (as *np.float64*)

- **ramp_rates** (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier for units, (as *str*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Return type
pd.DataFrame

nempy.time_sequential.**create_seed_ramp_rate_parameters**(*historical_dispatch*, *as_bid_ramp_rates*)
Combine historical dispatch and as bid ramp rates to get seed ramp rate parameters for a time sequential model.

Examples

```
>>> historical_dispatch = pd.DataFrame({
... 'unit': ['A', 'B'],
... 'initial_output': [80.0, 100.0]})
```

```
>>> as_bid_ramp_rates = pd.DataFrame({
... 'unit': ['A', 'B'],
... 'ramp_down_rate': [600.0, 1200.0],
... 'ramp_up_rate': [600.0, 1200.0]})
```

```
>>> create_seed_ramp_rate_parameters(historical_dispatch,
...                                  as_bid_ramp_rates)
  unit  initial_output  ramp_down_rate  ramp_up_rate
0    A            80.0           600.0         600.0
1    B           100.0          1200.0        1200.0
```

Parameters

- `historical_dispatch` (*pd.DataFrame*) –

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)

- `as_bid_ramp_rates` –

Columns:	Description:
unit	unique identifier for units, (as *str*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Returns

Columns:	Description:
unit	unique identifier for units, (as *str*)
initial_output	the output/consumption of the unit at the start of the dispatch interval, in MW, (as *np.float64*)
ramp_up_rate	the ramp up rate, in MW/h, (as *np.float64*)
ramp_down_rate	the ramp down rate, in MW/h, (as *np.float64*)

Return type
pd.DataFrame

CHAPTER

SEVEN

PUBLICATIONS

Links to publications and associate source code.

7.1 Nempy Technical Brief

The nempy technical brief is available `here`. as pdf, and is also used as the introduction for readthedocs page. The code below uses Nempy v1.1.0 which is now superseded v2.0.0.

7.1.1 Source code for Figure 1

```python
# Notice:
# - This script downloads large volumes of historical market data from AEMO's nemweb
#   portal. The boolean on line 20 can be changed to prevent this happening repeatedly
#   once the data has been downloaded.

import sqlite3
import pandas as pd
import random
from datetime import datetime, timedelta

from nempy import markets
from nempy.historical_inputs import loaders, mms_db, \
    xml_cache, units, demand, interconnectors, \
    constraints

# The size of historical data files for a full year of 5 min dispatch
# is very large, approximately 800 GB, for this reason the data is
# stored on an external SSD.
con = sqlite3.connect('F:/nempy_test_files/historical_mms.db')
mms_db_manager = mms_db.DBManager(connection=con)
xml_cache_manager = xml_cache.XMLCacheManager('F:/nempy_test_files/nemde_cache')

# The second time this example is run on a machine this flag can
# be set to false to save downloading the data again.
download_inputs = False

if download_inputs:
    mms_db_manager.populate(start_year=2019, start_month=1,
```

(continues on next page)

(continued from previous page)

```
                            end_year=2019, end_month=12)
    xml_cache_manager.populate(start_year=2019, start_month=1,
                               end_year=2020, end_month=1)

raw_inputs_loader = loaders.RawInputsLoader(
    nemde_xml_cache_manager=xml_cache_manager,
    market_management_system_database=mms_db_manager)


# Define a function for creating a list of randomly selected dispatch
# intervals
def get_test_intervals(number):
    start_time = datetime(year=2019, month=1, day=1, hour=0, minute=0)
    end_time = datetime(year=2019, month=12, day=31, hour=0, minute=0)
    difference = end_time - start_time
    difference_in_5_min_intervals = difference.days * 12 * 24
    random.seed(1)
    intervals = random.sample(range(1, difference_in_5_min_intervals), number)
    times = [start_time + timedelta(minutes=5 * i) for i in intervals]
    times_formatted = [t.isoformat().replace('T', ' ').replace('-', '/') for t in times]
    return times_formatted


# List for saving outputs to.
outputs = []

# Create and dispatch the spot market for each dispatch interval.
for interval in get_test_intervals(number=1000):
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    interconnector_inputs = interconnectors.InterconnectorData(raw_inputs_loader)
    constraint_inputs = constraints.ConstraintData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)

    # By default the CBC open source solver is used, but GUROBI is
    # also supported
    market.solver_name = 'CBC'  # or could be 'GUROBI'

    # Set bids
    volume_bids, price_bids = unit_inputs.get_processed_bids()
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    # Set bid in capacity limits
    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['unit_capacity']
```

(continues on next page)

(continued from previous page)

```
    market.make_constraints_elastic('unit_bid_capacity', violation_cost=cost)

    # Set limits provided by the unconstrained intermittent generation
    # forecasts. Primarily for wind and solar.
    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
    market.set_unconstrained_intermitent_generation_forecast_constraint(
        unit_uigf_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['uigf']
    market.make_constraints_elastic('uigf_capacity', violation_cost=cost)

    # Set unit ramp rates.
    ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch()
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.make_constraints_elastic('ramp_up', violation_cost=cost)
    market.make_constraints_elastic('ramp_down', violation_cost=cost)

    # Set unit FCAS trapezium constraints.
    unit_inputs.add_fcas_trapezium_constraints()
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_max_avail']
    fcas_availability = unit_inputs.get_fcas_max_availability()
    market.set_fcas_max_availability(fcas_availability)
    market.make_constraints_elastic('fcas_max_availability', cost)
    cost = constraint_inputs.get_constraint_violation_prices()['fcas_profile']
    regulation_trapeziums = unit_inputs.get_fcas_regulation_trapeziums()
    market.set_energy_and_regulation_capacity_constraints(regulation_trapeziums)
    market.make_constraints_elastic('energy_and_regulation_capacity', cost)
    scada_ramp_down_rates = unit_inputs.get_scada_ramp_down_rates_of_lower_reg_units()
    market.set_joint_ramping_constraints_lower_reg(scada_ramp_down_rates)
    market.make_constraints_elastic('joint_ramping_lower_reg', cost)
    scada_ramp_up_rates = unit_inputs.get_scada_ramp_up_rates_of_raise_reg_units()
    market.set_joint_ramping_constraints_raise_reg(scada_ramp_up_rates)
    market.make_constraints_elastic('joint_ramping_raise_reg', cost)
    contingency_trapeziums = unit_inputs.get_contingency_services()
    market.set_joint_capacity_constraints(contingency_trapeziums)
    market.make_constraints_elastic('joint_capacity', cost)

    # Set interconnector definitions, limits and loss models.
    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)

    # Add generic constraints and FCAS market constraints.
    fcas_requirements = constraint_inputs.get_fcas_requirements()
    market.set_fcas_requirements_constraints(fcas_requirements)
```

(continues on next page)

(continued from previous page)

```
    violation_costs = constraint_inputs.get_violation_costs()
    market.make_constraints_elastic('fcas', violation_cost=violation_costs)
    generic_rhs = constraint_inputs.get_rhs_and_type_excluding_regional_fcas_
↪constraints()
    market.set_generic_constraints(generic_rhs)
    market.make_constraints_elastic('generic', violation_cost=violation_costs)
    unit_generic_lhs = constraint_inputs.get_unit_lhs()
    market.link_units_to_generic_constraints(unit_generic_lhs)
    interconnector_generic_lhs = constraint_inputs.get_interconnector_lhs()
    market.link_interconnectors_to_generic_constraints(
        interconnector_generic_lhs)

    # Set the operational demand to be met by dispatch.
    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)
    # Get unit dispatch without fast start constraints and use it to
    # make fast start unit commitment decisions.
    market.dispatch()
    dispatch = market.get_unit_dispatch()
    fast_start_profiles = unit_inputs.get_fast_start_profiles_for_dispatch(dispatch)
    market.set_fast_start_constraints(fast_start_profiles)
    if 'fast_start' in market.get_constraint_set_names():
        cost = constraint_inputs.get_constraint_violation_prices()['fast_start']
        market.make_constraints_elastic('fast_start', violation_cost=cost)

    # If AEMO historical used the over constrained dispatch rerun
    # process then allow it to be used in dispatch. This is needed
    # because sometimes the conditions for over constrained dispatch
    # are present but the rerun process isn't used.
    if constraint_inputs.is_over_constrained_dispatch_rerun():
        market.dispatch(allow_over_constrained_dispatch_re_run=True,
                        energy_market_floor_price=-1000.0,
                        energy_market_ceiling_price=14500.0,
                        fcas_market_ceiling_price=1000.0)
    else:
        # The market price ceiling and floor are not needed here
        # because they are only used for the over constrained
        # dispatch rerun process.
        market.dispatch(allow_over_constrained_dispatch_re_run=False)

    # Save prices from this interval
    prices = market.get_energy_prices()
    prices['time'] = interval

    # Getting historical prices for comparison. Note, ROP price, which is
    # the regional reference node price before the application of any
    # price scaling by AEMO, is used for comparison.
    historical_prices = mms_db_manager.DISPATCHPRICE.get_data(interval)

    prices = pd.merge(prices, historical_prices,
                      left_on=['time', 'region'],
                      right_on=['SETTLEMENTDATE', 'REGIONID'])
```

(continues on next page)

(continued from previous page)

```python

    outputs.append(
        prices.loc[:, ['time', 'region', 'price',
                       'SETTLEMENTDATE', 'REGIONID', 'ROP']])

con.close()
outputs = pd.concat(outputs)
outputs = outputs.sort_values('ROP')
outputs = outputs.reset_index(drop=True)
outputs.to_csv('energy_price_results_2019_1000_intervals.csv')

```

7.1.2 Source code for Figure 2

```python
# Notice:
# - This script downloads large volumes of historical market data from AEMO's nemweb
#   portal. The boolean on line 20 can be changed to prevent this happening repeatedly
#   once the data has been downloaded.

import sqlite3
import pandas as pd
import random
from datetime import datetime, timedelta

from nempy import markets
from nempy.historical_inputs import loaders, mms_db, \
    xml_cache, units, demand, interconnectors, \
    constraints

# The size of historical data files for a full year of 5 min dispatch
# is very large, approximately 800 GB, for this reason the data is
# stored on an external SSD.
con = sqlite3.connect('historical_mms.db')
mms_db_manager = mms_db.DBManager(connection=con)
xml_cache_manager = xml_cache.XMLCacheManager('nemde_cache')

# The second time this example is run on a machine this flag can
# be set to false to save downloading the data again.
download_inputs = True

if download_inputs:
    # This requires approximately 5 GB of storage.
    mms_db_manager.populate(start_year=2019, start_month=1,
                            end_year=2019, end_month=12)

    # This requires approximately 3.5 GB of storage.
    xml_cache_manager.populate(start_year=2019, start_month=1, start_day=1,
                               end_year=2020, end_month=1, end_day=1)

raw_inputs_loader = loaders.RawInputsLoader(
```

(continues on next page)

(continued from previous page)

```
    nemde_xml_cache_manager=xml_cache_manager,
    market_management_system_database=mms_db_manager)


# Define a function for creating a list of randomly selected dispatch
# intervals
def get_test_intervals(number):
    start_time = datetime(year=2019, month=1, day=1, hour=0, minute=0)
    end_time = datetime(year=2019, month=12, day=31, hour=0, minute=0)
    difference = end_time - start_time
    difference_in_5_min_intervals = difference.days * 12 * 24
    random.seed(1)
    intervals = random.sample(range(1, difference_in_5_min_intervals), number)
    times = [start_time + timedelta(minutes=5 * i) for i in intervals]
    times_formatted = [t.isoformat().replace('T', ' ').replace('-', '/') for t in times]
    return times_formatted


# List for saving outputs to.
outputs = []

# Create and dispatch the spot market for each dispatch interval.
for interval in get_test_intervals(number=1000):
    raw_inputs_loader.set_interval(interval)
    unit_inputs = units.UnitData(raw_inputs_loader)
    interconnector_inputs = interconnectors.InterconnectorData(raw_inputs_loader)
    constraint_inputs = constraints.ConstraintData(raw_inputs_loader)
    demand_inputs = demand.DemandData(raw_inputs_loader)

    unit_info = unit_inputs.get_unit_info()
    market = markets.SpotMarket(market_regions=['QLD1', 'NSW1', 'VIC1',
                                                'SA1', 'TAS1'],
                                unit_info=unit_info)

    # By default the CBC open source solver is used, but GUROBI is
    # also supported
    market.solver_name = 'CBC'  # or could be 'GUROBI'

    # Set bids
    volume_bids, price_bids = unit_inputs.get_processed_bids()
    volume_bids = volume_bids[volume_bids['service'] == 'energy']
    price_bids = price_bids[price_bids['service'] == 'energy']
    market.set_unit_volume_bids(volume_bids)
    market.set_unit_price_bids(price_bids)

    # Set bid in capacity limits
    unit_bid_limit = unit_inputs.get_unit_bid_availability()
    market.set_unit_bid_capacity_constraints(unit_bid_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['unit_capacity']
    market.make_constraints_elastic('unit_bid_capacity', violation_cost=cost)

    # Set limits provided by the unconstrained intermittent generation
```

(continues on next page)

(continued from previous page)

```
    # forecasts. Primarily for wind and solar.
    unit_uigf_limit = unit_inputs.get_unit_uigf_limits()
    market.set_unconstrained_intermitent_generation_forecast_constraint(
        unit_uigf_limit)
    cost = constraint_inputs.get_constraint_violation_prices()['uigf']
    market.make_constraints_elastic('uigf_capacity', violation_cost=cost)

    # Set unit ramp rates.
    ramp_rates = unit_inputs.get_ramp_rates_used_for_energy_dispatch()
    market.set_unit_ramp_up_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_up_rate']])
    market.set_unit_ramp_down_constraints(
        ramp_rates.loc[:, ['unit', 'initial_output', 'ramp_down_rate']])
    cost = constraint_inputs.get_constraint_violation_prices()['ramp_rate']
    market.make_constraints_elastic('ramp_up', violation_cost=cost)
    market.make_constraints_elastic('ramp_down', violation_cost=cost)

    # Set interconnector definitions, limits and loss models.
    interconnectors_definitions = \
        interconnector_inputs.get_interconnector_definitions()
    loss_functions, interpolation_break_points = \
        interconnector_inputs.get_interconnector_loss_model()
    market.set_interconnectors(interconnectors_definitions)
    market.set_interconnector_losses(loss_functions,
                                     interpolation_break_points)

    # Set the operational demand to be met by dispatch.
    regional_demand = demand_inputs.get_operational_demand()
    market.set_demand_constraints(regional_demand)
    market.dispatch()

    # Save prices from this interval
    prices = market.get_energy_prices()
    prices['time'] = interval

    # Getting historical prices for comparison. Note, ROP price, which is
    # the regional reference node price before the application of any
    # price scaling by AEMO, is used for comparison.
    historical_prices = mms_db_manager.DISPATCHPRICE.get_data(interval)

    prices = pd.merge(prices, historical_prices,
                      left_on=['time', 'region'],
                      right_on=['SETTLEMENTDATE', 'REGIONID'])

    outputs.append(
        prices.loc[:, ['time', 'region', 'price',
                       'SETTLEMENTDATE', 'REGIONID', 'ROP']])

con.close()
outputs = pd.concat(outputs)
outputs = outputs.sort_values('ROP')
outputs = outputs.reset_index(drop=True)
```

(continues on next page)

(continued from previous page)

```
outputs.to_csv('energy_price_results_2019_1000_intervals_without_FCAS_or_generic_
↪constraints.csv')
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

I

U

W

X