Multi-period modeling in oemof.solph

Overview on implementation of MultiPeriodModels to enhance the oemof.solph framework

Outline

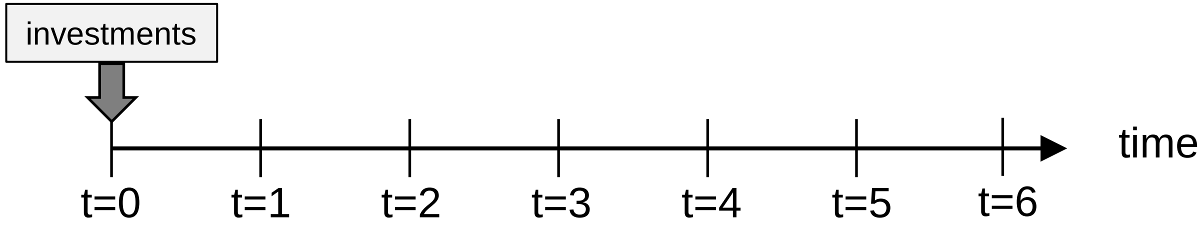
1. **Overview**

2. Description of changes by module

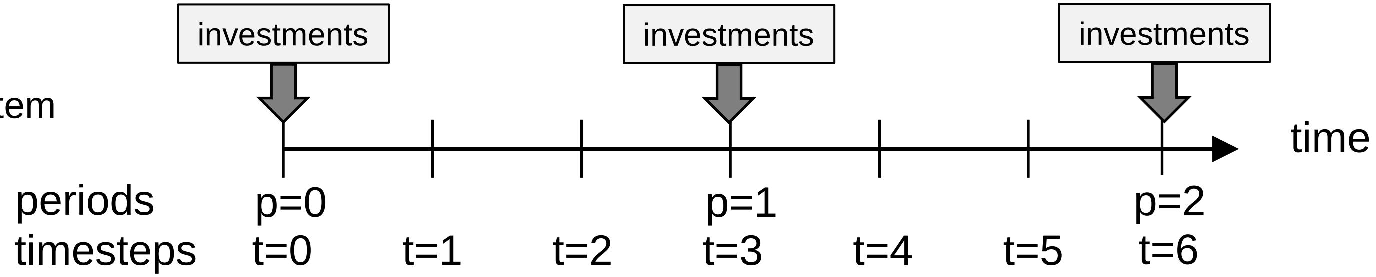
3. Description of logics used

Foundations: Multi-period optimization

- **Status quo** in oemof.solph
  - Only one timestep where investments may occur → t=0, i.e. the begin of the optimization
  - Investments are accounted for by their annuity
  - Timesteps are a one-dimensional set
  - Classical use case: Dimensioning of a system; short-term view (e.g. one typical year)



- Idea of **a multi-period optimization** model
  - Depict different periods in addition to different timesteps → there are two time-related indices
  - Investments may occur in every period
  - Some kind of lifetime tracking is needed
  - Classical use case Long-term planning of a system



Overview on changes by module (1/2)

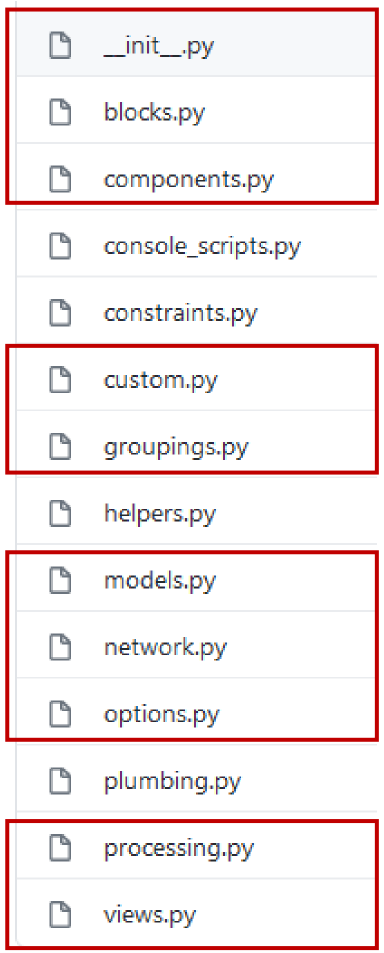
Modules touched

- __init__.py
- blocks.py
- components.py
- console_scripts.py
- constraints.py
- custom.py
- groupings.py
- helpers.py
- models.py
- network.py
- options.py
- plumbing.py
- processing.py
- views.py

Module	Change
__init__.py	Add imports of new classes: - oemof.solph.model.MultiPeriodModel - oemof.solph.options.MultiPeriod - oemof.solph.options.MultiPeriodInvestment
blocks.py	Add new classes: - MultiPeriodFlow - MultiPeriodInvestmentFlow - MultiPeriodBus - MultiPeriodTransformer
components.py	Add new classes: - GenericMultiPeriodStorageBlock - GenericMultiPeriodInvestmentStorageBlock
custom.py	Add new classes: - MultiPeriodLinkBlock - SinkDSMMultiPeriodBlocks - SinkDSMMultiPeriodInvestmentBlocks
groupings.py	Add groupings: - multiperiod_flow_grouping - multiperiodinvestment_flow_grouping
models.py	Add new class: - MultiPeriodModel

Overview on changes by module (2/2)

Modules touched



Module	Change
network.py	Add Flow attributes: - `multiperiod` - `multiperiodinvestment` - `fixed_costs` Add checks for attributes: - `multiperiod` - `multiperiodinvestment` → return MultiPeriodBlocks
options.py	Add new option: - MultiPeriodInvestment
processing.py	Extend results extraction: - Change processing.create_dataframe - Change processing.results - Change functions for timeindex extraction
views.py	Extend results extraction: - Change views.node (consequential amendment for changes in processing.py)

Basic principles & scope limitations

- In general, the focus has been on keeping **modularisation** and **reusage** by other modelers.

- **Extensions**
  - The **framework itself** has been **kept as is**.
  - The **multiperiod features** were added **on top**, building on what is already there.

- **Limitations**
  - For multiperiod modeling, **some sacrifice on generalizability** and abstraction had to be made.
    - It e. g. had to be determined, how to deal with discounting and using nominal or real values.
    - Periods are the years extracted from the timeindex. (This could be adapted.)

  - The focus for the enhancements has been on features to be used in the power market model POMMES.
    The most relevant components, but **not every component** has been prepared for usage in a MultiPeriodModel.
    - In the modules components.py and custom.py not everything has been touched.
    - The components touched besides Flows, Buses and Transformers, comprise GenericStorages, Links & SinkDSM units. Esp. the CHP components have not been touched (!).
    - The constraints.py module has not been touched (yet).

Outline

1. Overview
2. **Description of changes by module**
3. Description of logics used

models.py – class: MultiPeriodModel

```python
# periods equal to years (will probably be the standard use case)
periods = sorted(list(set(getattr(self.es.timeindex, 'year'))))
d = dict(zip(periods, range(len(periods))))

# pyomo set for timesteps of optimization problem
self.TIMESTEPS = po.Set(initialize=range(len(self.es.timeindex)),
                        ordered=True)
```

new

```python
self.TIMEINDEX = po.Set(
    initialize=list(zip([d[p] for p in self.es.timeindex.year],
                        range(len(self.es.timeindex)))),
    ordered=True)
```

new

```python
self.PERIODS = po.Set(initialize=range(len(periods)))
```

```python
def _add_parent_block_variables(self):
    """
    """
    self.flow = po.Var(self.FLOWS, self.TIMEINDEX,
                       within=po.Reals)
```

new

- Example: 3 years with 3 timesteps within each year

- Pyomo-Sets:
  - PERIODS: {0, 1, 2}
  - TIMESTEPS: {0, 1, 2, 3, 4, 5, 6, 7, 8}
  - TIMEINDEX: {(0, 0), (0, 1), (0, 2), (1, 3), (1, 4), (1, 5), (2, 6), (2, 7), (2, 8)}

same as in existing framework

tuples: (period, timestep)

- flow variable is defined over pyomo Set TIMEINDEX
- Iteration used: for p, t in self.TIMEINDEX
  - p: Periods
  - t: timesteps

- Not displayed: MultiPeriodModel itself carries the discount_rate information in an attribute `discount_rate`

(oemof.solph.)network.py – Flow attributes & checks for returning blocks

```python
scalars = ['nominal_value', 'summed_max', 'summed_min',
           'investment', 'multiperiod', 'multiperiodinvestment',
           'nonconvex', 'integer']
sequences = ['fix', 'variable_costs', 'fixed_costs', 'min', 'max']
```

new

Reintroduced (for multiperiod only)

- **Flow**
  - New attributes for multi-period modeling

```python
if 'fixed_costs' in keys:
    msg = ("Be aware that the fixed costs attribute is only\n"
           "meant to be used for MultiPeriodModels.\n"
           "It has been decided to remove the `fixed_costs` "
           "attribute with v0.2 for regular uses!")
    warn(msg, debugging.SuspiciousUsageWarning)
```

```python
def constraint_group(self):
    if self.balanced and not self.multiperiod:
        return blocks.Bus
    if self.balanced and self.multiperiod:
        return blocks.MultiPeriodBus
    else:
        return None
```

- **Bus**
  - Check for `multiperiod ` attribute and if it is True, return MultiPeriodBus

```python
# Check outputs for multiperiod modeling
for v in self.outputs.values():
    if (hasattr(v, 'multiperiod')
            or hasattr(v, 'multiperiodinvestment')):
        if (v.multiperiod is not None
                or v.multiperiodinvestment is not None):
            self.multiperiod = True
            break
        else:
            self.multiperiod = False

def constraint_group(self):
    if not self.multiperiod:
        return blocks.Transformer
    else:
        return blocks.MultiPeriodTransformer
```

- **Transformer**
  - Check for `multiperiod ` or `multiperiodinvestment` attribute in outflows (!) and if one of them is set, return MultiPeriodTransformers

* ***Notes***:
- *Existing capacities are treated as sunk costs and not considered*
- *If the lifetime is larger than the optimization horizon, the costs until the end of the lifetime are added anyways, in order not to falsely incentivize investments occuring close to the end of he optimization horizon.*

blocks.py – New Flow classes

```python
class MultiPeriodFlow(SimpleBlock):
    r""" Block for all flows with :attr:`multiperiod` being not None.
```

```python
class MultiPeriodInvestmentFlow(SimpleBlock):
    r"""Block for all flows with :attr:`multiperiodinvestment` being not None.

        # create invest variable for a multiperiodinvestment flow
        self.invest = Var(self.MULTIPERIODINVESTFLOWS,
                          m.PERIODS,
                          within=NonNegativeReals,
                          bounds=_investvar_bound_rule)

        # Total capacity
        self.total = Var(self.MULTIPERIODINVESTFLOWS,
                         m.PERIODS,
                         within=NonNegativeReals)

        # Old capacity to be decommissioned (due to lifetime)
        self.old = Var(self.MULTIPERIODINVESTFLOWS,
                       m.PERIODS,
                       within=NonNegativeReals)
```

- **MultiPeriodFlow**
  - Pretty much the same as standard dispatch flow for usage in a MultiPeriodModel
  - Indexation by TIMEINDEX (period, timestep)

- **MultiPeriodInvestmentFlow**
  - Similar to InvestmentFlow, but for usage in a MultiPeriodModel
  - **New variables**
    - invest: invested capacity
    - **total**: installed capacity after decommissionings
    - **old**: capacity to be decommissioned due to exceeding its lifetime
  - **New constraints**
    - lifetime tracking and decommissioning
    - overall_maximum: impose an limit on overall installation for all periods
    - overall_minimum: define a minimum that has to be installed in the last period
  - **Adjusted objective value**
    - discounted variable costs (same for MultiPeriodFlow)
    - annuity of CAPEX and fixed costs **for the lifetime****

blocks.py – New Bus and Transformer classes

```
class MultiPeriodBus(SimpleBlock):
    r"""Block for all balanced MultiPeriodBuses.
```

```
class MultiPeriodTransformer(SimpleBlock):
    r"""Block for the linear relation of nodes with type
    :class:`~oemof.solph.network.Transformer` used if :attr:`multiperiod` or
    :attr:`multiperiodinvestment` is True
```

- **MultiPeriodBus** & **MultiPeriodTransformer**
  - Pretty much the same as standard components, but for usage in a MultiPeriodModel
  - Indexation of flow vars by TIMEINDEX (period, timestep) in the constraints formulation

* existing capacities are treated as sunk costs and not considered

components.py – New Storage Blocks

```python
def constraint_group(self):
    if self._invest_group is True:
        return GenericInvestmentStorageBlock
    elif self._multiperiodinvest_group is True:
        return GenericMultiPeriodInvestmentStorageBlock
    elif self._invest_group is False and not self.multiperiod:
        return GenericStorageBlock
    elif self._multiperiodinvest_group is False and self.multiperiod:
        return GenericMultiPeriodStorageBlock
    else:
        e = (
            "Infeasible combination of attributes\n"
            "Won't return any constraints block for GenericStorage."
        )
        raise AttributeError(e)
```

```python
class GenericMultiPeriodStorageBlock(SimpleBlock):
    r"""Storage without an :class:`.MultiPeriodInvestment` object.
```

```python
class GenericMultiPeriodInvestmentStorageBlock(SimpleBlock):
    r"""
    Block for all storages with :attr:`MultiPeriodInvestment` being not None.
    See :class:`oemof.solph.options.MultiPeriodInvestment` for all parameters
    of the MultiPeriodInvestment class.
```

- **GenericStorage.constraint_group**
  - Check which attributes are set and determine which Block to return

- **GenericMultiPeriodStorageBlock**
  - Pretty much the same as standard GenericStorageBlock, but for usage in a MultiPeriodModel
  - Indexation of flow vars by TIMEINDEX (period, timestep)
  - discounted fixed costs are included in the _objective_expression

- **GenericMultiPeriodStorageInvestmentBlock**
  - Based on GenericInvestmentStorageBlock
  - Similar to MultiPeriodInvestmentFlow → new vars, lifetime tracking, objective value calculation
  - **No initial_storage_level** (resp. set to 0) → I found it hard to interpret any other value
  - discounted annuities of CAPEX as well as fixed costs for the lifetime are included in the _objective_expression

* ***Notes***:
- *There is a pending PR on that (#740). I'll resume with that ASAP and afterwards adjust the multieriod(investment) implementation.*

custom.py – New Link and SinkDSM Blocks

```
def constraint_group(self):
    if not self.multiperiod:
        return LinkBlock
    else:
        return MultiPeriodLinkBlock
```

```
class MultiPeriodLinkBlock(SimpleBlock):
    r"""Block for the relation of nodes with type
    :class:`~oemof.solph.custom.Link` with the :attr:`multiperiod`
```

```
if self.approach == possible_approaches[0]:
    if self._invest_group is True:
        return SinkDSMDIWInvestmentBlock
    elif self._multiperiodinvest_group is True:
        return SinkDSMDIWMultiPeriodInvestmentBlock
    elif self.multiperiod == True:
        return SinkDSMDIWMultiPeriodBlock
    else:
        return SinkDSMDIWBlock
```

```
class SinkDSMDLRMultiPeriodBlock(SimpleBlock):
    r"""Constraints for SinkDSMDLRBlock
```

```
class SinkDSMDLRMultiPeriodInvestmentBlock(SinkDSMDLRBlock):
    r"""Constraints for SinkDSMDLRInvestmentBlock
```

- **Link.constraint_group**
  - Check which attributes are set and determine which Block to return
- **MultiPeriodLinkBlock**
  - Nothing special, just flow indexation by TIMEINDEX (period, timestep)
- **SinkDSM***
  - Extensive implementation of different approaches Check for attributes and return respective block
- **SinkDSMMultiPeriodBlock(s)***
  - A dispatch only version: adjustments are the same as for MultiPeriodFlow
- **SinkDSMMultiPeriodInvestmentBlock(s)***
  - An investment verion: adjustments are the same as for MultiPeriodInvestmentFlow

groupings.py – New groupings

```python
def _multiperiod_grouping(stf):
    if hasattr(stf[2], 'multiperiod'):
        if stf[2].multiperiod is not None:
            return True
    else:
        return False


multiperiod_flow_grouping = groupings.FlowsWithNodes(
    constant_key=blocks.MultiPeriodFlow,
    # stf: a tuple consisting of (source, target, flow), so stf[2] is the flow.
    filter=_multiperiod_grouping)


def _multiperiodinvestment_grouping(stf):
    if hasattr(stf[2], 'multiperiodinvestment'):
        if stf[2].multiperiodinvestment is not None:
            return True
    else:
        return False


multiperiodinvestment_flow_grouping = groupings.FlowsWithNodes(
    constant_key=blocks.MultiPeriodInvestmentFlow,
    # stf: a tuple consisting of (source, target, flow), so stf[2] is the flow.
    filter=_multiperiodinvestment_grouping)
```

- **multiperiod_flow_grouping**
  - Group flows with attribute `multiperiod`
- **multiperiodinvestment_flow_grouping**
  - Group flows with attribute `multiperiodinvestment`

processing.py – adapted results extraction

* ***Notes***:
- *Some vars are indexed by timesteps as in the standard model, others by periods or by the timeindex (tuple of period, timestep).*
- *This has to be taken into account when mapping the results back since pyomo doesn‘t deliver that information.*
- *There is room for improving this particular implementation in terms of performance and coding style.*

- Functions get_timeindex and remove_timeindex replace get_timestep and remove_timestep for multiperiod models

```python
def get_timeindex(x):
    """
    Get the timeindex from oemof tuples for multiperiod models.
    Slice int values (timeindex, timesteps or periods) dependent on how
    the variable is indexed.

    The timestep is removed from tuples of type `(n, n, int, int)`,
    `(n, n, int)` and `(n, int)`.
    """
    for i, n in enumerate(x):
        if isinstance(n, int):
            return x[i:]
    else:
        return (0,)
```

```python
def remove_timeindex(x):
    """
    Remove the timeindex from oemof tuples for mulitperiod models.
    Slice up to integer values (node labels)

    The timestep is removed from tuples of type `(n, n, int, int)`,
    `(n, n, int)` and `(n, int)`.
    """
    for i, n in enumerate(x):
        if isinstance(n, int):
            return x[:i]
    else:
        return x
```

```python
if isinstance(om, models.MultiPeriodModel):
    # Note: timeindex differs dependent on variables!
    period_indexed = ['invest', 'total', 'old']
    period_timestep_indexed = ['flow']
    # TODO: Take care of initial storage content instead of just ignoring
    to_be_ignored = ['init_content']
    timestep_indexed = [el for el in df['variable_name'].unique()
                        if el not in period_indexed
                        and el not in period_timestep_indexed
                        and el not in to_be_ignored]
    time_col = 'timeindex'
    scalars_col = 'period_scalars'
```

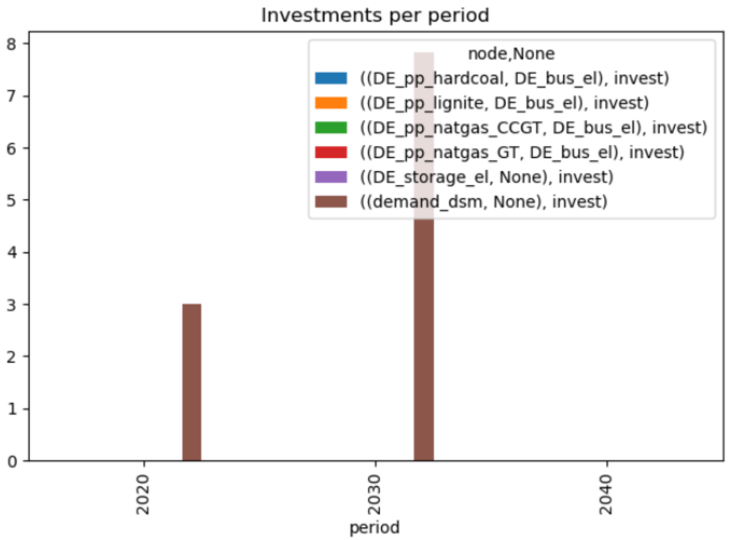
- **processing.results**
  - Check if model is a MultiPeriodModel
  - Distinct the different indexing of vars*
  - Extract results and map them back to the timeindex

views.py – adapted results extraction

* ***Notes***:
- *Some vars are indexed by timesteps as in the standard model, others by periods or by the timeindex (tuple of period, timestep).*
- *This has to be taken into account when mapping the results back since pyomo doesn‘t deliver that information.*
- *There is room for improving this particular implementation in terms of performance and coding style.*

```
scalars_col = 'scalars'
# Check for MultiPeriodModel (different naming)
if 'period_scalars' in list(list(results.values())[0].keys()):
    scalars_col = 'period_scalars'
```

- **views.node**
  - Check if model is a MultiPeriodModel → model is not in memory; hence check for adjusted naming introduced for a MultiPeriodModel
  - Take the correct values for proper results extraction → period_scalars has an investment value for every period



- **Results visualization for a toy model (outside the framework)**
  - investments are spread over different periods
  - results extraction is not so straightforward yet, though

Outline

1. Overview
2. Description of changes by module
3. **Description of logics used**

Lifetime logic

- P: installed capacity
- p: period
- n: lifetime

$$P_{total}(p) = P_{invest}(p) + P_{total}(p-1) - P_{old}(p) \; \forall p > 0$$

total cap: previous cap + installations - decommissionings

$$P_{total}(p) = P_{invest}(p) + P_{existing} \; \forall p = 0$$

$$P_{old}(p) = P_{invest}(p-n) \; \forall p > n$$

decomissionings: installations that happened in the period the plants lifetime ago

$$P_{old}(p) = P_{existing} + Pinvest(0) \; \forall p = n - age$$

$$P_{old}(p) = 0 \; else$$

Handling cost values (1/2)

- In general: all cost values may vary on a **periodical basis**, but shall be fixed within a period.

- Cost values have to be provided **in nominal terms**.
    - Calculating real values and annuities takes place under the hood.

- **Annuities and discounting**
    - A **discount_rate** is given on a **model-wide basis**. It reflects inflation.
    - An **interest rate may be given per component / flow** (asset) that can be invested in. It can deviate from the discount_rate, e.g. to take an investor‘s view and demand for higher interests. If a social planner perspective is taken, the interest_rate should be equal to the model‘s discount_rate, which is the default.
    - Annuities are calculated under the hood (next slide).

Handling cost values (2/2)

- Cost terms for MultiPeriodInvestment objects (or other components that is invested in)

CAPEX: investment annuities

$$P_{invest}(p) \cdot annuity(c_{invest}, n, i)(p) \cdot n \cdot DF(p) \, \forall p$$

$$annuity(c_{invest}, n, i) = \frac{(1+i)^n \cdot i}{(1+i)^n - 1} \cdot c_{invest}$$

Fixed costs

$$\sum_p P_{invest}(p) \cdot c_{fixed}(p) \cdot DF(pp) \, \forall pp \in [p, p+n] * DF(p)$$

with discount factor

$$DF(p) = (1+d)^{-p}$$

- P: installed capacity
- p: period
- n: lifetime
- i: interest rate (may deviate from discount rate)
- d: discount rate
- DF: discount factor

Open points & Outlook

- Missing multi-period implementation of components in components & custom.
- Missing tests so far.
  - Testing has been made by gradually adjusting the toy model that is provided among the changes.
  - Model (blocks) pprints have been thoroughly inspected.
  - Target function values have been plausibilized by recalculating them.
- Some minor documentation update yet to follow.
- The import statements in some modules were changed. This has to be reset again. The toy model is kept within the framework package. It will be removed.
- Outlook
  - One main aim was to provide a working new feature for the community that may be used with care by experienced users.
  - Feedback on the new feature is highly appreciated.
  - Further modifications and enhancements of the new feature by the community are highly encouraged.

Thank you!

- A warm thank you goes out to the oemof-dev community for their fruitful advice on this feature.
- A special thank you goes out to Simon Hilpert who pretty much did the basic work which was built upon.
- Another special thank you goes to Johannes Giehl, Yannick Werner and Benjamin Grosse for their advice on certain implementation issues.