Multi-period modeling in oemof.solph

(J. Kochems & J. Giehl)

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
4. Discussion

Why multi-period optimization?

- The standard oemof.solph configuration **does fit** for modelling **short time horizons**
  - A classic use case would be the investigation of one year for an energy system
  - There is a rather detailed representation of the technology application / dispatch
  - This is suited only for investment decisions at the beginning of the time horizon (comparison if new technology can be cheaper)

- Current investment approach with only one period **does <u>not</u> fit** to long **long-term scenarios**
  - There is only one investment timestep at the beginning of the modelling horizon (t=0) which is rather unrealistic
  - This leads to modelling the whole horizon under present (fix) conditions
    → *Paradigm: You build your energy system today which is still in place in 2050, or worse 2100*

- One idea is tackling this shortcoming with a **multi-period** approach
  - In reality, an investor would rather replace assets after their lifetime on a periodical basis
  - This allows for the integration of intertemporal physical and financial flow decisions

Advantages of a (perfect foresight) multi-period model

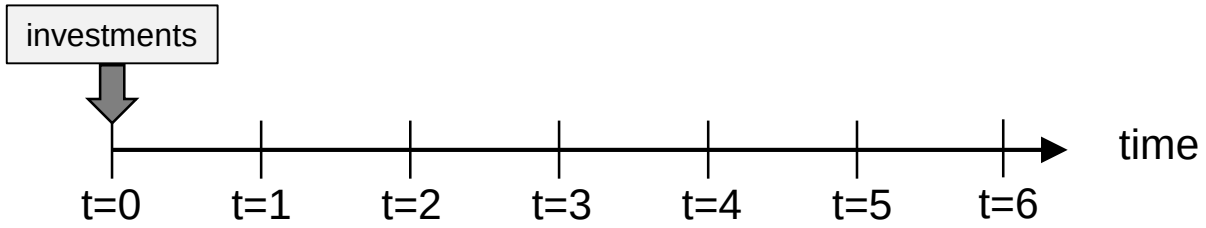
- Extended model does not loose the **overall, intertemporal optima**
  - Model can determine the periods in which capacities should be switched off or installed
  - It allows for an adequate consideration of cost degression (or increases) over time
  - Total system costs can be summed up from the discounted values over all periods

- Doing so, it allows to derive **long-term system scenarios** from a "**social planner**" perspective
  - There is no sequential solution of the problem necessary (e.g. myopic approach)
  - The approach becomes especially relevant to optimize the long-term investments of an energy system
  - One major assumption is a foreseeable future under perfect information

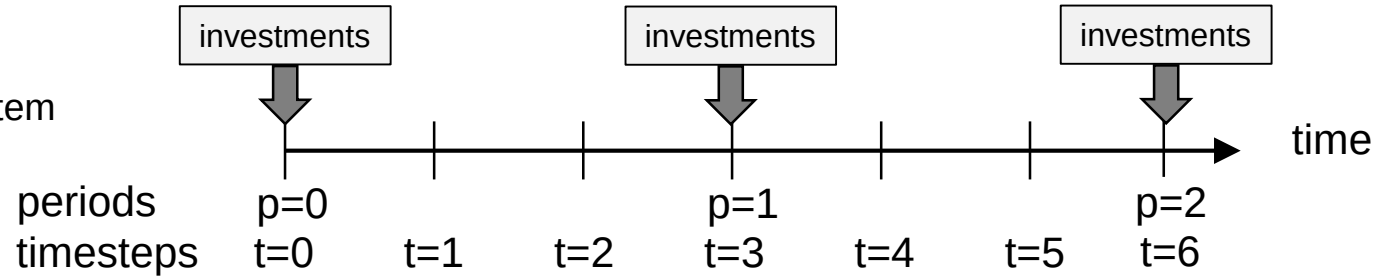
- With some modifications and additions, **microeconomic models could in principle be build as well**
  - Approach models the investment behavior closer to realistic situations where re-investments and shutdowns are an option
  - Parametric uncertainties may be provided by some sort of forecasting model or stochastic programming (which could also potentially benefit from the implementation itself)
  - Some sort of consideration of expected revenues could be added in order to decide on plant shutdowns

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)

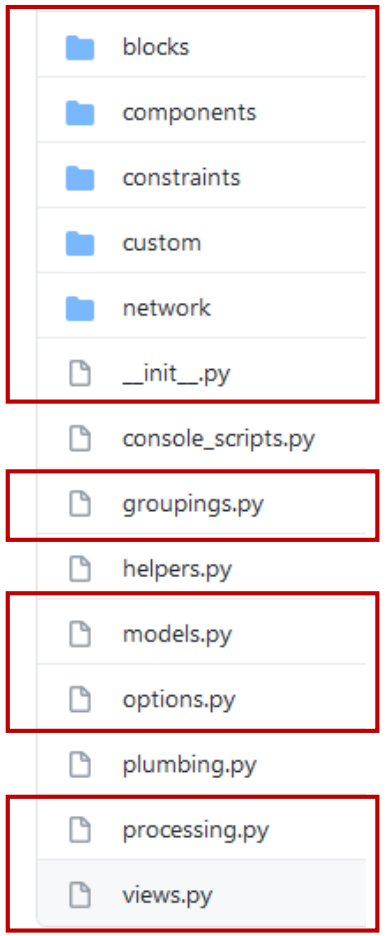


- Idea of **a multi-period optimization** model
  - Depict different periods in addition to different timesteps → there are two time-related indices
  - Investments may occur in every period
  - Some kind of lifetime tracking is needed
  - Classical use case: long-term planning of a system



Overview on changes by module (1/2)

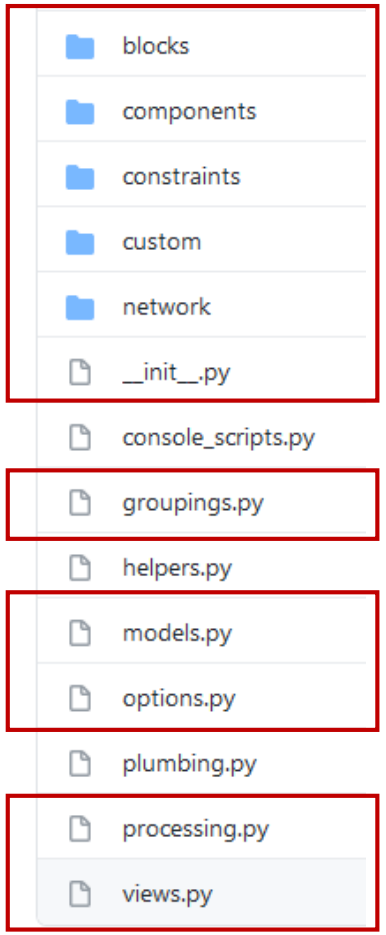
Modules / Packages touched



Module / package	Change
Package blocks	Add new modules and classes: - multiperiod_flow; class: MultiPeriodFlow - multiperiodinvestment_flow; class: MultiPeriodInvestmentFlow - multiperiod_bus; class: MultiPeriodBus - multiperiod_transformer; class: MultiPeriodTransformer
Package components	Add new classes: - generic_storage.GenericMultiPeriodStorageBlock - generic_storage.GenericMultiPeriodInvestmentStorageBlock
Package constraints	Add changes: - integral_limit for MultiPeriodModel - new module multiperiodinvestment_limit.py
Package custom	Add changes: - new class MultiPeriodLinkBlock in link.py - new class SinkDSMMultiPeriodBlocks in sink_dsm.py - SinkDSMMultiPeriodInvestmentBlocks in sink_dsm.py
Package network	Add Flow attributes (in flow.py): - `multiperiod` - `multiperiodinvestment` - `fixed_costs` Add checks for attributes (in bus.py, transformer.py: - `multiperiod` - `multiperiodinvestment` → return MultiPeriodBlocks

Overview on changes by module (2/2)

Modules / Packages touched



Module	Change
oemof/solph/__init__.py	Add imports of new classes: - oemof.solph.model.MultiPeriodModel - oemof.solph.options.MultiPeriodInvestment
groupings.py	Add groupings: - multiperiod_flow_grouping - multiperiodinvestment_flow_grouping
models.py	Add new class: - new class MultiPeriodModel
options.py	Add new option: - MultiPeriodInvestment (similar to options.Investment, but with overall_maximum, overall_minimum, lifetime, fixed_costs, interest_rate, …)
processing.py	Extend results extraction: - Change function processing.create_dataframe - Change function processing.results (handle indices) - Add / change functions for timeindex extraction
views.py	Extend results extraction: - Change function 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 fully been adapted (yet).

Outline

1. Overview

2. **Description of changes by module**

3. Description of logics used

4. Discussion

models.py – class: MultiPeriodModel

```
# 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

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

new

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

```
def _add_parent_block_variables(self):
    """
    """
```

new

```
    self.flow = po.Var(self.FLOWS, self.TIMEINDEX,
                       within=po.Reals)
```

- 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 package – Flow attributes & checks for returning blocks

```
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)

- **class Flow in flow.py**
  - New attributes for multi-period modeling

```
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)
```

```
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
```

- **class Bus in bus.py**
  - Check for `multiperiod ` attribute and if it is True, return MultiPeriodBus

```
    # 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
```

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

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

blocks package – New Flow classes

* ***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 the optimization horizon.*

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

```
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)
        # Old capacity is built out of old exogenous and endogenous capacities
        self.old = Var(self.MULTIPERIODINVESTFLOWS,
                       m.PERIODS,
                       within=NonNegativeReals)
```

- **class MultiPeriodFlow in multiperiod_flow.py**
  - Pretty much the same as standard dispatch flow for usage in a MultiPeriodModel
  - Indexation by TIMEINDEX (period, timestep)
- **class MultiPeriodInvestmentFlow in multiperiodinvestment_flow.py**
  - 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; in turn consists of exogeneous and endogeneous capacity
  - **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 package – 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
```

- **Class MultiPeriodBus in multiperiod_bus.py & class MultiPeriodTransformer in multiperiod_transformer.py**
  - 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

components package – New Storage Blocks in generic_storage.py

```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***:
- *Changes from PR (#740) will be remerged. Multiperiod implementation will be tidied up.*

Custom package – 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 is 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

```
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, i.e. a 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'
```

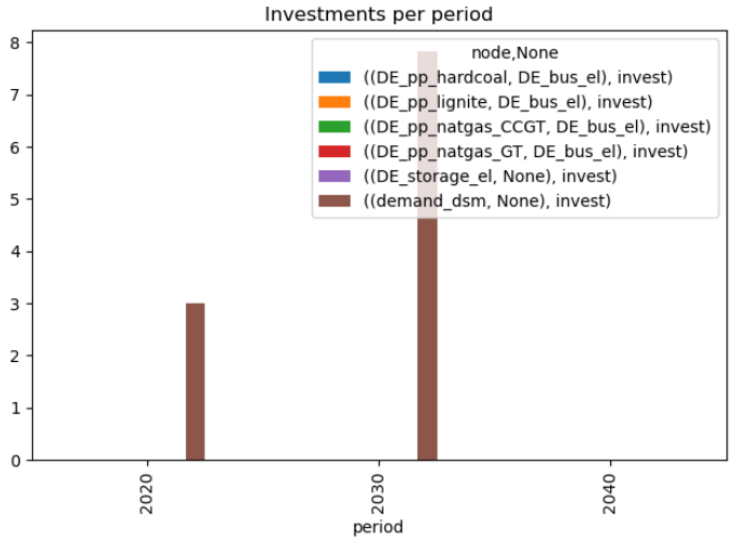
- **processing.results**
  - Check if model is a MultiPeriodModel
  - Distinct the different indexing of vars*
  - Extract results and map them back to the timeindex

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

constraints package - integral_limit and multiperiodinvestment_limit.py

```python
if isinstance(om, MultiPeriodModel):
    setattr(
        om,
        limit_name,
        po.Expression(
            expr=sum(
                om.flow[inflow, outflow, p, t]
                * om.timeincrement[t]
                * sequence(getattr(flows[inflow, outflow], keyword))[t]
                for (inflow, outflow) in flows
                for p, t in om.TIMEINDEX
            )
        ),
    )
```

- **Changes for integral_limit.py**
  - 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

```python
def multiperiodinvestment_rule(m):
    expr = 0

    if hasattr(m, "MultiPeriodInvestmentFlow"):
        expr += m.MultiPeriodInvestmentFlow.investment_costs

    if hasattr(m, "GenericMultiPeriodInvestmentStorageBlock"):
        expr += m.GenericMultiPeriodInvestmentStorageBlock.investment_costs

    return expr <= limit

model.multiperiodinvestment_limit = po.Constraint(
    rule=multiperiodinvestment_rule)
```

- **New module multiperiodinvestment_limit.py**
  - Pretty straightforward and similar to regular model; Limit on overall investment costs
  - Sum up investment_costs and make sure they are smaller or equal to limit

options.py – New class MultiPeriodInvestment

```
def __init__(self, maximum=float('+inf'), minimum=0, ep_costs=0,
             existing=0, nonconvex=False, offset=0,
             overall_maximum=None, overall_minimum=None,
             lifetime=0, age=0, interest_rate=0, fixed_costs=None,
             **kwargs):
```

```
    def _check_age_and_lifetime(self):
        if self.age >= self.lifetime:
            e4 = ("A unit's age must be smaller than its "
                  "expected lifetime.")
            raise AttributeError(e4)
```

- **class MultiPeriodInvestment**
  - standard parameters from options.Investment (maximum, minimum, ep_costs, existing, nonconvex, offset)
  - Lifetime tracking related parameters (lifetime, age)
  - Other parameters (overall_maximum, overall_minimum, interest_rate, fixed_costs)
  - some check routines

Outline

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

Lifetime logic

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

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

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

$$P_{old,end}(p) = P_{invest}(p-n) \quad \forall p \geq n$$

$$P_{old,end}(p) = 0 \quad else$$

$$P_{old,exo}(p) = P_{existing} \quad \forall p = n - age$$

$$P_{old,exo}(p) = 0 \quad else$$

$$P_{old}(p) = P_{old,end}(p) + P_{old,exo}(p)$$

total (installed) cap: previous cap + installations - decommissionings

Decomissionings

- endogeneous plants: installations that happened in the period the plants lifetime ago
- exogeneous plants: decommissioning of existing capacity in period lifetime – (initial) age
- Total decommissioning: sum of endogeneous and exogeneous decommissioning

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 interest rates.
    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

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

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

- P: installed capacity
- p: period
- n: lifetime
- i: interest rate
- DF: discount factor

Fixed costs

$$\sum_{pp=p}^{p+n} P_{invest}(p) \cdot c_{fixed}(pp) \cdot DF(pp) \cdot DF(p) \; \forall p \in PERIODS$$

with discount factor

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

Open points & Outlook

- Missing multi-period implementation of components in **components** & **custom** and some **constraints**.

- **Missing tests** / test updates 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.

- A **toy model** has been used for testing the framework changes. 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 to Johannes Giehl who developped parts of this feature.
- A special thank you goes out to Simon Hilpert who pretty much did the basic work which was built upon.
- A warm thank you goes out to the oemof-dev community for their fruitful advice on this feature, esp. Patrik Schönfeldt and Uwe Krien.
- Another special thank you goes to Yannick Werner and Benjamin Grosse for their advice on certain implementation issues.

Outline

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

Aspects for discussion

- Timeline
  - Rough idea is to get it done by May / June
  - Progress depends on other tasks and maybe a bit on support and requirements from the community

- Who is willing to support on this one?
  - Help would be appriciated
    - Code reviews & hints
    - For developping certain model features, such as CHP components and other custom components

- How strongly to generalize and coordinate with other needs?
  - The approach sacrifices on generality.
    - There are some options to increase it.
    - For other aspects, taking decisions seems reasonable (such as discounting and handling fixed costs).
  - Some synergies with multi-scenario modeling where a second timeindex is needed are there.
    - A decision to touch every variable would deserve further attention.
    - As of now, the approach rather checks which varables have to be touched and tries to keep as much untouched as possible.

Indexation issue

- Description
  - Indexation differs by vars type and is taken into account in processing.py
  - Advantage: Less code changes necessary
  - Drawback: Filtering for var types needed

- Vars indexation
  - Vars indexed by periods: investment, total, old, old_exo, old_end
  - Vars indexed by timeindex: flow
  - Ignored vars (ignored in processing): init_content (storage)
  - Timesteps indexed (as before)
    - storage_content
    - dsm_do_shed, dsm_do_level, dsm_up_level, dsm_do_shift, dsm_up, balance_dsm_do, balance_dsm_up (DSM vars; here: DLR approach)

Current state of implementation

- oemof.solph (draft) Pull Request #750
  - Almost all necessary modules touched
  - Last commit already roughly a month ago, but activity to follow (always hard to coordinate my freetime PhD activity ;-))
  - Some failing tests and CI checks need attention
  - Next up: easy fix: merge dev back and include changes for SinkDSM in order to add a proper multiperiod implementation for that and at least fix the PEP8 issues

- Overview on changes by module
  - https://github.com/oemof/oemof-solph/pull/750/files
  - Touches 25 files in total (some of which are new)

Contact

Johannes Kochems

research associate at DLR, Institute of Networked Energy Systems, Stuttgart |
PhD candidate at Technical University of Berlin

E-Mail:
johannes.kochems@dlr.de
kochems@campus.tu-berlin.de

GitHub: jokochems