Hydropower_in_pypsa

April 21, 2023

1 Hydropower constraint in PyPSA with Linopy

This notebook uses a toy PyPSA model to showcase how hydropower dispatch can be bounded by the historical hydro operation. With this approach, we avoid that the bulk hydropower reservoir capacity is operated in a too ideal manner caused by the year-ahead perfect foresight assumed in the model.

The toy model is a copper plate representation of the Norwegian electricity system. Hydropower capacity is included exogenously (Norway has a capacity of 37.7 GW) while wind and solar generation capacity are included endogenously, i.e., they are subject to optimization. Furthermore, as an electricity storage option, we make Li-Ion batteries available to be invested in.

Capital costs used in this toy model do not represent real values but are solely used as exemplary data. The same applies to the considered scenario. The scenario is realized by assuming that the electricity load is increased by a factor 2 to force a capacity expansion. Secondly, we force the system to rely heavily on solar to introduce a large mismatch during winter. In this period, the perfect foresight of the model entails that hydropower reservoirs are discharged heavily, while the dispatch during summer is very low. To ensure that we do not interrupt with other non-energy system phenomena, e.g., water accesability and ecosystem flow continuity requirements, we need to constrain the dispatch.

For a robust PyPSA-Eur implementation, the following things need to be made: - Write the query code that fetches the ENTSO-E dispatch data (currently, this was manually downloaded and the recent years are for this reason not included) - Create a Github repo that stores all historical dispatch data (it would be inconvenient if this was added directly to the PyPSA-Eur repo) - Extend this jupyter notebook to account for multiple nodes within a country, i.e., historical data needs to be split and scaled (according to population/demand?)

The capacity factors of solar and wind are acquired from the course in Renewable Energy Systems taught by Marta Victoria.

Thanks for the suggestions by Fabian Hofmann on how to use Linopy to define this constraint, discussed here.

2 Table of content

2.1 - Historical dispatch

2.2 - Build toy model

2.3 - Unconstrained hydro

2.4 - Constrained hydro

Import useful modules:

[1]:
```
import pypsa
import pandas as pd
import numpy as np
import xarray as xr
import importlib
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import yaml
import warnings
warnings.filterwarnings("ignore")
```

The following lines ensure continuous reloading of functions which is convenient in the editing stage:

[2]:
```
%load_ext autoreload
%autoreload 2
```

Import functions:

[3]:
```
from plotting import plot_layout, plot_historical_dispatch,␣
 ↪plot_hydro_operation, plot_electricity_supply, plot_total_electricity_supply
from investment import annuity, build_base_network #, solve_network
```

Standardize figure layout:

[4]:
```
fs = 18
plot_layout(fs)
tech_colors_path = 'tech_colors.yaml'
with open(tech_colors_path) as file:
    tech_colors = yaml.safe_load(file)['tech_colors']
```

Pick a country (here, we consider Norway due to their high share of hydropower):

[5]:
```
country = 'NOR'

country_iso_alpha2 = {'NOR':'NO'} # country codes
long_country_name = {'NOR':'Norway',
                     'SWE':'Sweden',
                     'AUT':'Austria'}
hydro_cap = {'NOR':37732} # the aggregate power capacity of each countries
```

```
hydro_max_hours = {'NOR':1129.7} # the max hours (duration), which implicitly␣
↪defines the energy capacity, of the hydro reservoir
```

3 Historical inflow

[6]:
```
inflow_values = pd.read_csv('data/inflow/Hydro_inflow_' +␣
↪country_iso_alpha2[country] + '.csv', sep=',', index_col=0)
hydro_dates = pd.date_range('1/1/2003','31/12/2012',freq='d')
inflow_series = pd.Series(index=hydro_dates,
                          data=inflow_values['Inflow [GWh]'].values)*1e3 #␣
↪convert GWh to MWh
inflow_series_hourly = inflow_series.resample(rule='h').mean().
↪interpolate(method='linear')/24
```

Available years:

[7]:
```
print('Historical inflow years: ', list(inflow_series_hourly.index.year.
↪unique()))
```

```
Historical inflow years:  [2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011,
2012]
```

Pick one of the year contained in the above list:

[8]:
```
hydro_year = 2011
inflow_series_hourly_year = inflow_series_hourly.loc[inflow_series_hourly.
↪index[inflow_series_hourly.index.year == hydro_year]]
```

4 Historical dispatch

The question is, do we want to limit the hourly power generation or the aggregate energy production? We test this in the following.

Read historical data on hydro reservoir dispatch:

[9]:
```
historical_dispatch = pd.read_csv('ENTSOE_data/dispatch_' +␣
↪long_country_name[country] + '_2015-2018.csv',index_col=0)
historical_dispatch.index = pd.to_datetime(historical_dispatch.index)
df = pd.DataFrame(index=pd.date_range('1/1/2015','1/1/
↪2016',freq='h',closed='left'))
```

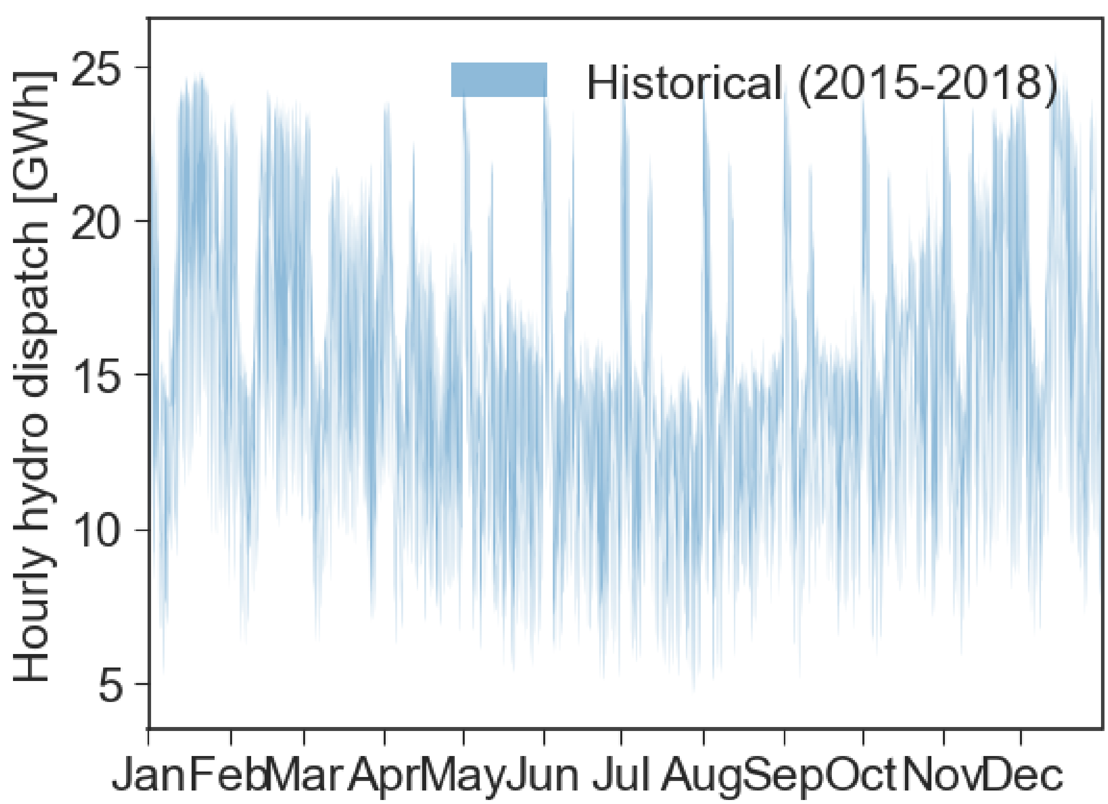
Split data by year:

[10]:
```
df1 = pd.DataFrame(historical_dispatch)
df1['year'] = df1.index.year
for year in df1.index.year.unique():
    df1_year = df1.query('year == @year')
    df1_year = df1_year[~df1_year.index.duplicated(keep='first')]
```

```
    if (year % 4 == 0) and (year % 100 != 0): # leap year
        day_29 = df1_year.index.day == 29
        feb_29 = df1_year[day_29][df1_year[day_29].index.month == 2]
        df1_year = df1_year.drop(index=feb_29.index)

    df[year] = df1_year.disp.values
```

4.1 Hourly dispatch

[11]:
```
df_min = df.min(axis=1)
df_max = df.max(axis=1)

fig, ax = plt.subplots()
x = df_min.index
y1 = df_min
y2 = df_max
ax.fill_between(x,y1,y2,alpha=0.5,label='Historical (2015-2018)')

ax.set_xlim(min(x),max(x))
ax.legend()
ax.set_ylabel('Hourly hydro dispatch [GWh]')

fmt = mdates.DateFormatter('%b')
ax.xaxis.set_major_formatter(fmt);
```



We do see a seasonal trend. However, note that peak dispatch power is observed troughout the year. Althouh low dispatch is observed on average during summer, hours with full utilization of the hydropower capacity is still observed in this period. From this observation, we learn that **we should not constrain the hourly dispatch**.

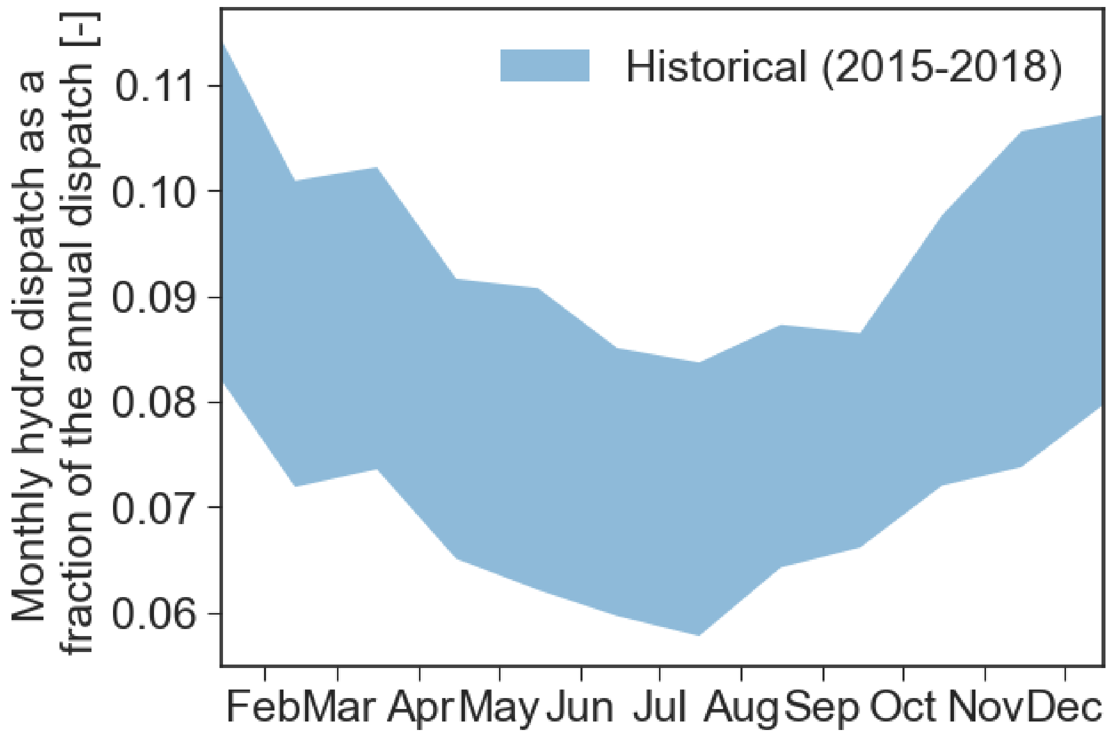
We can check this for other countries (to do so, uncomment the below line), here exemplified with Sweden:

[12]:
```
# plot_historical_dispatch('SWE',long_country_name,freq='h')
```

4.2 Monthly aggregate dispatch

Instead of looking at hourly dispatch, let's consider the monthly aggregate. Here, we illustrate the historical band of the dispatch based on monthly aggregate:

[13]:
```
df_min = df.min(axis=1)
df_max = df.max(axis=1)

fig, ax = plt.subplots()
x = df_min.resample('m').sum().index - pd.Timedelta('15d')
y1 = df_min.resample('m').sum()/(df.mean(axis=1).resample('m').sum().sum())
y2 = df_max.resample('m').sum()/(df.mean(axis=1).resample('m').sum().sum())

ax.fill_between(x,y1,y2,alpha=0.5,label='Historical (2015-2018)')
ax.set_xlim(min(x),max(x))
ax.legend()
ax.set_ylabel('Monthly hydro dispatch as a \n fraction of the annual dispatch␣
 ↪[-]')

fmt = mdates.DateFormatter('%b')
ax.xaxis.set_major_formatter(fmt);
```



By using the monthly aggregate, we can constrain the hydropower more realistically to retain the historical seasonality - and we still allow hours with peak power production during summer.

5 Investment optimization

5.1 Create network

Initialize network:

[14]:
```
n = pypsa.Network()
```

Add buses (here, an electricity bus):

[15]:
```
n.add("Bus","electricity bus")
```

Add carriers:

[16]:
```
carriers = ['wind','solar','hydro','battery']
n.madd("Carrier",carriers);
```

Add time snapshots:

[17]:
```
hours = pd.date_range('2015-01-01T00:00Z','2015-12-31T23:00Z', freq='H')
n.set_snapshots(hours)
```

Add load:

Let's assume the case at which Norway increses its electricity demand by a factor of 2. Otherwise, no capacity expansion would take place since hydro can cover all the load.

---

[18]:
```
load_csv = 'data/electricity_demand.csv'
load = pd.read_csv(load_csv, sep=';', index_col=0)
load.index = pd.to_datetime(load.index)
load_scale_up = 2
n.add('Load', # pypsa component
      'el_load', # name
      bus = 'electricity bus', # bus name
      p_set = load[country]*load_scale_up
     )
```

6 Add generators and storage

Let's force the system to have more solar than wind. In this way, without a hydropower constraint, the system chooses to have high hydro dispatch during winter to overcome the concurrently low availability of solar energy. As mentioned in the introduction, this does not represent any real scenario but is more a "model trick" to showcase an example.

You can tweek these numbers *p_nom_max_wind* and *p_nom_max_solar* to change scenarios between solar or wind-dominant systems

---

[19]:
```
p_nom_wind  = 0
p_nom_solar = 0
p_nom_max_wind = 10e3 # 10e10
p_nom_max_solar = 10e5 # 10e10
```

6.1 *Wind* (added endogenously)

Capacity factors:

[20]:
```
df_onshorewind = pd.read_csv('data_extra/onshore_wind_1979-2017.csv', sep=';',␣
 ↪index_col=0)
df_onshorewind.index = pd.to_datetime(df_onshorewind.index)
CF_wind = df_onshorewind[country][[hour.strftime("%Y-%m-%dT%H:%M:%SZ") for hour␣
 ↪in n.snapshots]]
```

Add generator:

[21]:
```
n.add('Generator',
      'wind',
      bus='electricity bus',
```

```
      carrier='wind',
      p_nom = p_nom_wind,
      p_nom_extendable=True,
      p_nom_max=p_nom_max_wind, # maximum capacity can be limited due to␣
 ↪environmental constraints
      capital_cost=1e6*annuity(30,0.07),
      marginal_cost=0.015,
      p_max_pu=CF_wind,
     )
```

6.2 *Solar* (added endogenously)

Capacity factors:

[22]:
```
df_solar = pd.read_csv('data_extra/pv_optimal.csv', sep=';', index_col=0)
df_solar.index = pd.to_datetime(df_solar.index)
CF_solar = df_solar[country][[hour.strftime("%Y-%m-%dT%H:%M:%SZ") for hour in n.
 ↪snapshots]]
```

Add generator:

[23]:
```
n.add('Generator',
      'solar',
      bus='electricity bus',
      carrier='solar',
      p_nom = p_nom_solar,
      p_nom_extendable=True,
      p_nom_max=p_nom_max_solar, # maximum capacity can be limited due to␣
 ↪environmental constraints
      capital_cost=1e5*annuity(30,0.07),
      marginal_cost=0.01,
      p_max_pu=CF_solar,
     )
```

6.3 *Hydro reservoir* (added exogenously)

[24]:
```
inflow_series_hourly_year.index = n.snapshots
hydro_inflow = inflow_series_hourly_year
```

[25]:
```
n.add('StorageUnit',
      'hydro',
      bus='electricity bus',
      carrier = 'hydro',
      p_nom_extendable=False,
      p_nom=hydro_cap[country],
      inflow=hydro_inflow,
      max_hours=hydro_max_hours[country],
      capital_cost=10e6*annuity(30,0.07),
```

```
        marginal_cost=0.,
        p_max_pu=1,
        p_min_pu=0,
        efficiency_dispatch=0.9,
        efficiency_store=0.0, # you can't store electricity in this item
        cyclic_state_of_charge=True,
       )
```

6.4 *Li-Ion battery* (added endogenously)

[26]:
```
n.add('StorageUnit',
      'battery',
      bus='electricity bus',
      carrier = 'battery',
      p_nom_extendable=True,
      capital_cost=1e6*annuity(30,0.07),
      max_hours=6,
      efficiency_store=0.95,
      efficiency_dispatch=0.95,
      cyclic_state_of_charge=True
     )
```

7 Results for an unconstrained hydro operation

It is a good idea to check if the network is built consistently. This can be done with the PyPSA function "consistency_check". From the PyPSA documentation:

"Checks the network for consistency; e.g. that all components are connected to existing buses and that no impedances are singular.

Prints warnings if anything is potentially inconsistent."

[27]:
```
n.consistency_check()
```

No output means that no inconsistencies are present. Them, we can proceed solving the network:

[28]:
```
def extra_functionality(n,snapshot):
    """
    Collects supplementary constraints which will be passed to
    ``pypsa.optimization.optimize``.
    If you want to enforce additional custom constraints, this is a good
    location to add them.
    """

    # we do not include any extra functionalities
```

[29]:
```
def solve_network(n,
                  **kwargs):
```

```python
import logging
import pypsa

logger = logging.getLogger(__name__)
pypsa.pf.logger.setLevel(logging.WARNING)


solver_options = {'threads': 4,
                  'method': 2, # barrier
                  'crossover': 0,
                  'BarConvTol': 1.e-6,
                  'Seed': 123,
                  'AggFill': 0,
                  'PreDual': 0,
                  'GURO_PAR_BARDENSETHRESH': 200,
                  'seed': 10}

cf_solving = {'formulation': 'kirchhoff',
              'clip_p_max_pu': 1.e-2,
              'load_shedding': False,
              'noisy_costs': True,
              'skip_iterations': True,
              'track_iterations': False,
              'min_iterations': 4,
              'max_iterations': 6,
              'seed': 123}

solver_name = 'gurobi'

track_iterations = cf_solving.get("track_iterations", False)
min_iterations = cf_solving.get("min_iterations", 4)
max_iterations = cf_solving.get("max_iterations", 6)

skip_iterations = cf_solving.get("skip_iterations", False)
if not n.lines.s_nom_extendable.any():
    skip_iterations = True
    logger.info("No expandable lines found. Skipping iterative solving.")

if skip_iterations:
    status, condition = n.optimize(solver_name=solver_name,
                                   extra_functionality=extra_functionality,
                                   **solver_options,
                                   **kwargs,
    )
else:
```

```
        status, condition = n.optimize.
↪optimize_transmission_expansion_iteratively(
            solver_name=solver_name,
            track_iterations=track_iterations,
            min_iterations=min_iterations,
            max_iterations=max_iterations,
            extra_functionality=extra_functionality,
            **solver_options,
            **kwargs,
        )

    if status != "ok":
        logger.warning(
            f"Solving status '{status}' with termination condition␣
↪'{condition}'"
        )
    if "infeasible" in condition:
        raise RuntimeError("Solving status 'infeasible'")

    return n
```

[30]:
```
solve_network(n)
```

```
INFO:__main__:No expandable lines found. Skipping iterative solving.
INFO:linopy.model: Solve linear problem using Gurobi solver
INFO:linopy.model:Solver options:
 - threads: 4
 - method: 2
 - crossover: 0
 - BarConvTol: 1e-06
 - Seed: 123
 - AggFill: 0
 - PreDual: 0
 - GURO_PAR_BARDENSETHRESH: 200
 - seed: 10
Writing constraints.:
100%|                                 | 20/20
[00:01<00:00, 18.46it/s]
Writing variables.:
100%|                                   | 7/7
[00:00<00:00, 31.32it/s]

Set parameter Username
Academic license - for non-commercial use only - expires 2024-03-22


Read LP format model from file C:\Users\au485969\AppData\Local\Temp\linopy-
problem-i3hzid2y.lp
```

```
Reading time = 0.48 seconds
obj: 166445 rows, 78843 columns, 302854 nonzeros
Set parameter Threads to value 4
Set parameter Method to value 2
Set parameter Crossover to value 0
Set parameter BarConvTol to value 1e-06
Set parameter Seed to value 123
Set parameter AggFill to value 0
Set parameter PreDual to value 0
Set parameter GURO_PAR_BARDENSETHRESH to value 200
Set parameter Seed to value 10
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, instruction set
[SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 4 threads

Optimize a model with 166445 rows, 78843 columns and 302854 nonzeros
Model fingerprint: 0x9a5e71fe
Coefficient statistics:
  Matrix range     [1e-03, 6e+00]
  Objective range  [1e-02, 8e+04]
  Bounds range     [4e+03, 4e+04]
  RHS range        [4e+03, 4e+07]
Presolve removed 100116 rows and 12511 columns
Presolve time: 0.16s
Presolved: 66329 rows, 66332 columns, 190227 nonzeros
Ordering time: 0.02s

Barrier statistics:
 Dense cols : 3
 AA' NZ     : 1.414e+05
 Factor NZ  : 7.730e+05 (roughly 60 MB of memory)
 Factor Ops : 9.471e+06 (less than 1 second per iteration)
 Threads    : 4

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   3.08464385e+11 -2.67015989e+15  3.45e+06 1.05e+01  3.05e+10     0s
   1   4.69294249e+11 -6.65128904e+14  3.59e+06 1.03e+03  9.80e+09     0s
   2   1.85296036e+11 -2.77864747e+14  1.90e+06 4.06e+02  5.22e+09     0s
   3   2.75400946e+11 -5.94221443e+13  4.20e+05 7.14e+01  9.83e+08     0s
   4   2.81922814e+11 -1.23724207e+13  1.34e+05 9.62e+00  2.39e+08     1s
   5   2.13913058e+11 -4.22223437e+12  3.21e+04 1.85e+00  6.18e+07     1s
   6   1.49507019e+11 -2.34844094e+12  1.30e+04 7.93e-01  2.86e+07     1s
   7   1.15095151e+11 -8.39938551e+11  6.79e+03 1.68e-01  1.06e+07     1s
   8   5.65813578e+10 -3.37440193e+11  1.87e+03 8.38e-03  3.62e+06     1s
   9   3.94904731e+10 -1.44480529e+11  1.10e+03 3.20e-10  1.58e+06     1s
```

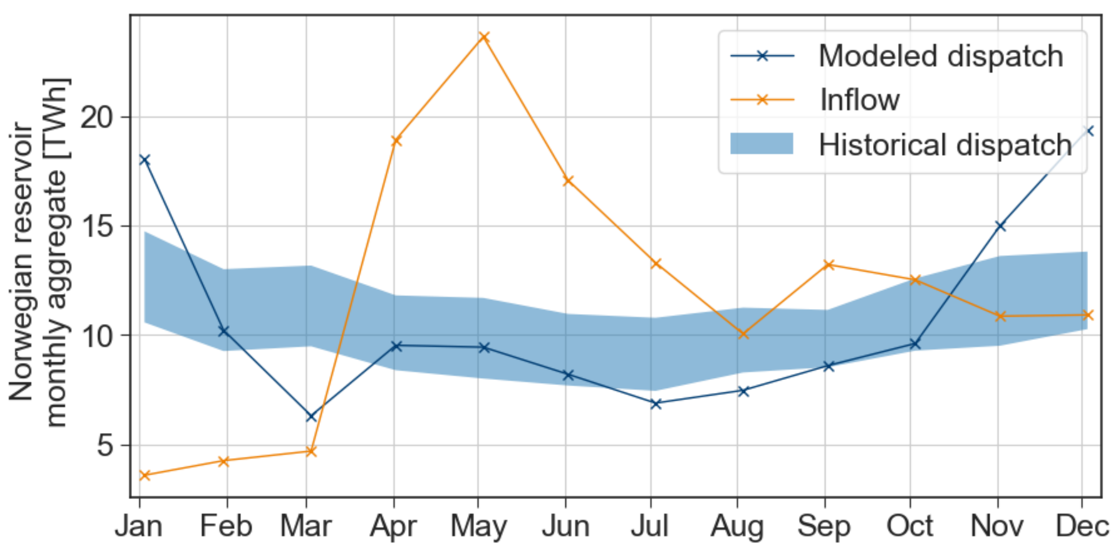
```
  10   3.02485451e+10 -1.39281859e+11  7.46e+02 3.20e-10  1.40e+06     1s
  11   2.62579979e+10 -1.02911687e+11  5.87e+02 7.96e-13  1.05e+06     1s
  12   2.45759973e+10 -9.39135666e+10  5.15e+02 7.96e-13  9.56e+05     1s
  13   2.40006281e+10 -8.28364126e+10  4.95e+02 7.96e-13  8.60e+05     1s
  14   2.29740172e+10 -7.26349626e+10  4.58e+02 7.96e-13  7.66e+05     1s
  15   2.19114493e+10 -6.98484159e+10  4.23e+02 6.82e-13  7.32e+05     1s
  16   2.12587621e+10 -6.37001164e+10  3.96e+02 6.82e-13  6.76e+05     1s
  17   2.04786825e+10 -5.84267953e+10  3.67e+02 6.25e-13  6.25e+05     1s
  18   1.96202132e+10 -5.51583161e+10  3.35e+02 6.25e-13  5.90e+05     1s
  19   1.92483150e+10 -4.51019416e+10  3.11e+02 9.31e-10  5.06e+05     1s
  20   1.67206900e+10 -3.71929790e+10  2.25e+02 3.98e-13  4.20e+05     1s
  21   1.38273648e+10 -2.34666899e+10  1.62e+02 2.84e-13  2.88e+05     1s
  22   1.22674117e+10 -7.06926932e+09  1.27e+02 1.56e-13  1.49e+05     1s
  23   1.08276011e+10 -6.68708376e+08  9.51e+01 1.69e-09  8.81e+04     2s
  24   1.00135526e+10  2.56753436e+09  7.52e+01 8.53e-14  5.71e+04     2s
  25   8.37952102e+09  4.11948464e+09  3.85e+01 1.75e-11  3.26e+04     2s
  26   7.96395793e+09  4.92765595e+09  2.41e+01 7.17e-11  2.32e+04     2s
  27   7.81543984e+09  5.54167670e+09  1.97e+01 7.62e-14  1.74e+04     2s
  28   7.59222716e+09  5.97712598e+09  1.27e+01 6.98e-10  1.23e+04     2s
  29   7.47532472e+09  6.20161101e+09  8.54e+00 5.07e-12  9.70e+03     2s
  30   7.41913852e+09  6.36177461e+09  6.00e+00 1.72e-09  8.04e+03     2s
  31   7.35619063e+09  6.74841886e+09  3.51e+00 6.65e-14  4.62e+03     2s
  32   7.31890363e+09  6.95428498e+09  2.17e+00 6.22e-11  2.77e+03     2s
  33   7.29904177e+09  7.04625052e+09  1.46e+00 4.66e-10  1.92e+03     2s
  34   7.28757151e+09  7.10998679e+09  1.03e+00 3.20e-10  1.35e+03     2s
  35   7.28132915e+09  7.13289419e+09  7.78e-01 2.91e-11  1.13e+03     2s
  36   7.27579342e+09  7.21080266e+09  5.54e-01 4.29e-11  4.96e+02     2s
  37   7.26976693e+09  7.24267445e+09  2.99e-01 4.37e-10  2.08e+02     2s
  38   7.26615595e+09  7.24746327e+09  1.39e-01 1.28e-09  1.43e+02     2s
  39   7.26298827e+09  7.25591689e+09  5.79e-03 1.46e-08  5.34e+01     2s
  40   7.26272086e+09  7.26154143e+09  7.34e-05 9.44e-08  8.91e+00     2s
  41   7.26264053e+09  7.26230799e+09  1.39e-05 2.74e-07  2.61e+00     2s
  42   7.26260975e+09  7.26243283e+09  4.15e-06 3.48e-07  1.45e+00     2s
  43   7.26260233e+09  7.26251647e+09  2.91e-06 3.69e-07  8.11e-01     2s
  44   7.26259105e+09  7.26256397e+09  1.35e-06 4.31e-07  4.08e-01     3s
  45   7.26258520e+09  7.26258612e+09  6.85e-07 4.55e-07  2.23e-01     3s

Barrier solved model in 45 iterations and 2.66 seconds (1.35 work units)
Optimal objective 7.26258520e+09


INFO:linopy.constants: Optimization successful:
Status: ok
Termination condition: optimal
Solution: 78843 primals, 166445 duals
Objective: 7.26e+09
Solver model: available
Solver message: 2
```

[30]:
```
PyPSA Network
Components:
 - Bus: 1
 - Carrier: 4
 - Generator: 2
 - Load: 1
 - StorageUnit: 2
Snapshots: 8760
```

[31]:
```
model_monthly_dispatch = n.storage_units_t.p['hydro'].resample('m').sum()/1e3
historical_monthly_dispatch_min = df.resample('m').sum().min(axis=1)
historical_monthly_dispatch_max = df.resample('m').sum().max(axis=1)
```

Hydropower operation:

[32]:
```
plot_hydro_operation(n,
                     df,
                     hydro_inflow,
                     freq='m'
                    )
```

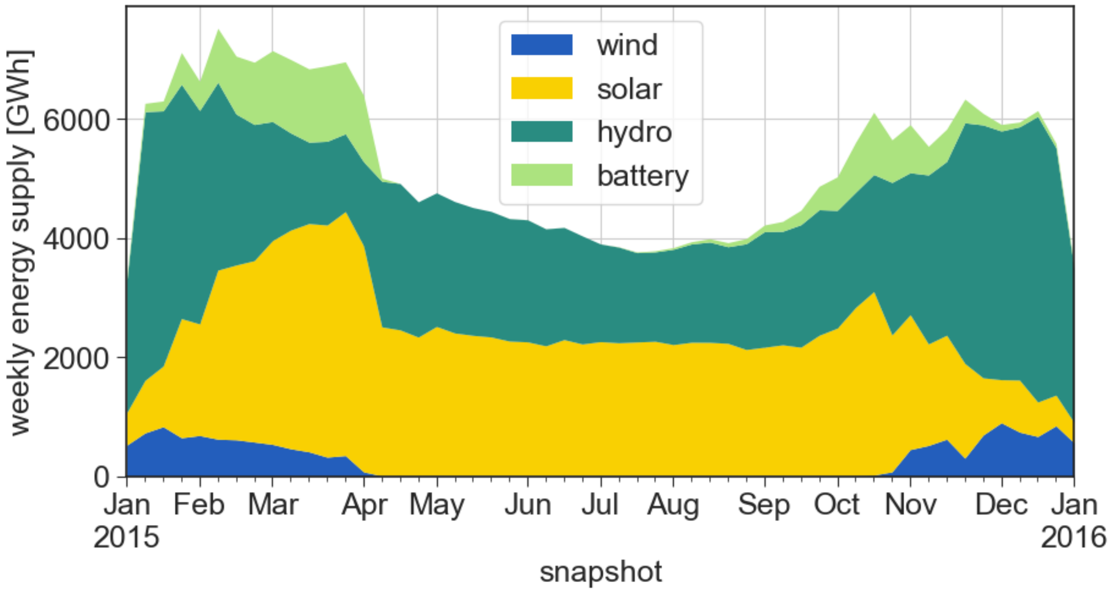


At a high share of solar, the modeled dispatch exceeds the historical region. This is in line with the findings in Gøtske and Victoria (2022).

Wind and solar power supply + battery balance:

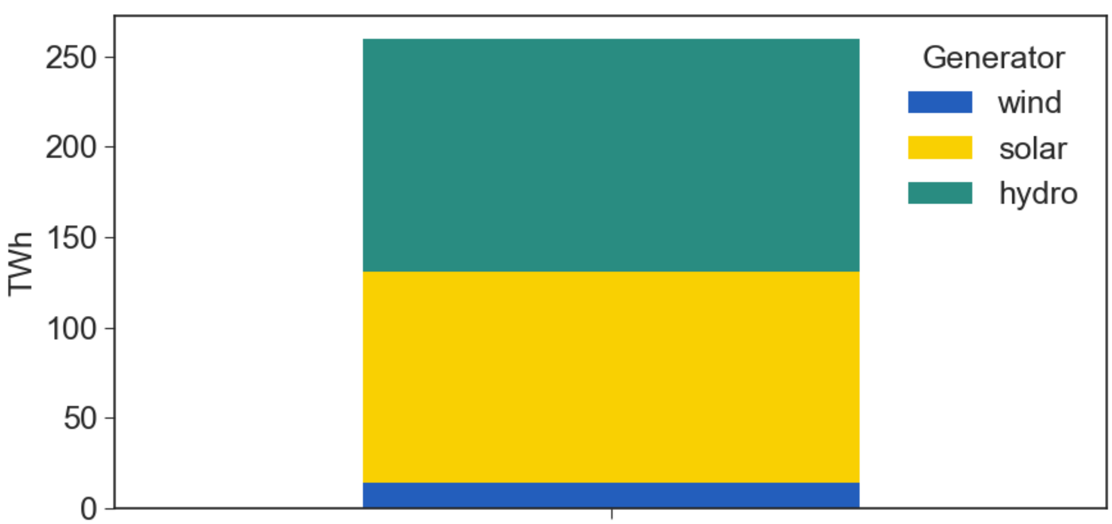
[33]:
```
plot_electricity_supply(n,
                        tech_colors,
```

```
                     freq='w')
```



Annual electricity supply:

[34]:
```
plot_total_electricity_supply(n,tech_colors)
```



8 Results for a constrained hydro operation

[35]:
```
n_c = build_base_network(country,
                         load_csv,
                         load_scale_up,
                         CF_wind,
                         p_nom_wind,
                         p_nom_max_wind,
                         CF_solar,
                         p_nom_solar,
                         p_nom_max_solar,
                         p_nom_hydro=hydro_cap[country],
                         hydro_inflow=hydro_inflow,
                         hydro_max_hours=hydro_max_hours[country],
                         carriers_list=['wind','solar','battery','hydro'])
```

[36]:
```
def add_upper_hydropower_constraint(n_c):
    """
    This function adds the upper bound constraint on the hydropower dispatch.
    """

    # LHS
    p = n_c.model.variables["StorageUnit-p_dispatch"].sel(StorageUnit='hydro')
    # p.groupby("snapshot.month").sum() # use this command instead when␣
↪implementing in PyPSA-Eur
    ds_months = pd.Series(n_c.snapshots.month,
                          index = pd.DatetimeIndex(n_c.snapshots)
                         ).to_xarray()
    lhs = p.groupby(ds_months).sum()

    # RHS
    snapshot = np.arange(1,13)
    limit = list(df_max.resample('m').sum()*1e3) # Assigning historical upper␣
↪limit
    data_array = xr.DataArray(
                data=limit,
                dims = ["snapshot"],
                coords = dict(snapshot=snapshot),
                attrs = dict(description="monthly_limit",units="")
                )
    rhs = data_array
    n_c.model.add_constraints(lhs <= rhs, name="hydro monthly upper bound")
```

[37]:
```
def add_lower_hydropower_constraint(n_c):
    """
    This function adds the lower bound constraint on the hydropower dispatch.
    """
```

```python
    # LHS
    p = n_c.model.variables["StorageUnit-p_dispatch"].sel(StorageUnit='hydro')
    # p.groupby("snapshot.month").sum() # use this command instead when␣
↪implementing in PyPSA-Eur
    ds_months = pd.Series(n_c.snapshots.month,
                          index = pd.DatetimeIndex(n_c.snapshots)
                         ).to_xarray()
    lhs = p.groupby(ds_months).sum()

    # RHS
    snapshot = np.arange(1,13)
    limit = list(df_min.resample('m').sum()*1e3) # Assigning historical lower␣
↪limit
    data_array = xr.DataArray(
                data=limit,
                dims = ["snapshot"],
                coords = dict(snapshot=snapshot),
                attrs = dict(description="monthly_limit",units="")
                )
    rhs = data_array
    n_c.model.add_constraints(lhs >= rhs, name="hydro monthly lower bound")
```

[38]:
```python
def extra_functionality(n_c,snapshot):
    """
    Collects supplementary constraints which will be passed to
    ``pypsa.optimization.optimize``.
    If you want to enforce additional custom constraints, this is a good
    location to add them.
    """

    add_lower_hydropower_constraint(n_c)
    add_upper_hydropower_constraint(n_c)
```

[39]:
```python
solve_network(n_c)
```

```
INFO:__main__:No expandable lines found. Skipping iterative solving.
INFO:linopy.model: Solve linear problem using Gurobi solver
INFO:linopy.model:Solver options:
 - threads: 4
 - method: 2
 - crossover: 0
 - BarConvTol: 1e-06
 - Seed: 123
 - AggFill: 0
 - PreDual: 0
 - GURO_PAR_BARDENSETHRESH: 200
 - seed: 10
```

```
Writing constraints.:
100%|                                | 22/22
[00:01<00:00, 21.51it/s]
Writing variables.:
100%|                                  | 7/7
[00:00<00:00, 60.39it/s]

Read LP format model from file C:\Users\au485969\AppData\Local\Temp\linopy-
problem-wgquqjb7.lp
Reading time = 0.35 seconds
obj: 166469 rows, 78843 columns, 320374 nonzeros
Set parameter Threads to value 4
Set parameter Method to value 2
Set parameter Crossover to value 0
Set parameter BarConvTol to value 1e-06
Set parameter Seed to value 123
Set parameter AggFill to value 0
Set parameter PreDual to value 0
Set parameter GURO_PAR_BARDENSETHRESH to value 200
Set parameter Seed to value 10
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, instruction set
[SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 4 threads

Optimize a model with 166469 rows, 78843 columns and 320374 nonzeros
Model fingerprint: 0x2b62efe7
Coefficient statistics:
  Matrix range     [1e-03, 6e+00]
  Objective range  [1e-02, 8e+04]
  Bounds range     [4e+03, 4e+04]
  RHS range        [4e+03, 4e+07]
Presolve removed 100128 rows and 12511 columns
Presolve time: 0.17s
Presolved: 66341 rows, 66344 columns, 198999 nonzeros
Ordering time: 0.02s

Barrier statistics:
 Dense cols : 3
 AA' NZ     : 1.589e+05
 Factor NZ  : 9.202e+05 (roughly 60 MB of memory)
 Factor Ops : 1.545e+07 (less than 1 second per iteration)
 Threads    : 4

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   8.70348473e+11 -2.75624082e+15  2.37e+09 9.01e+00  4.46e+10     0s
```

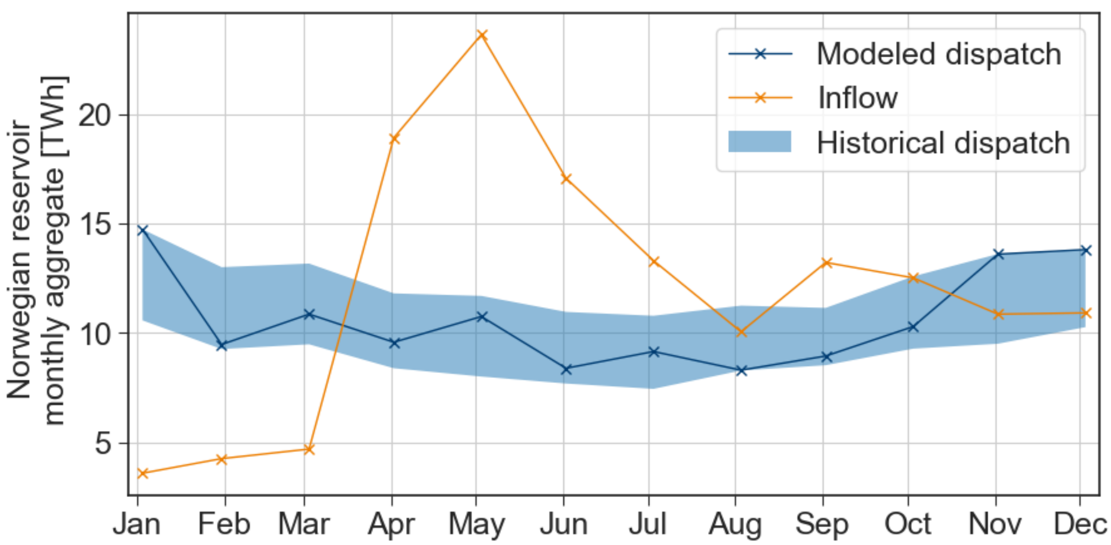
```
   1   1.30157645e+12 -6.71349496e+14  1.74e+09 7.74e+02  1.50e+10     0s
   2   5.19189504e+11 -2.79873216e+14  1.39e+09 2.84e+02  9.70e+09     0s
   3   7.38151654e+11 -6.64133888e+13  2.45e+08 5.11e+01  1.83e+09     1s
   4   6.93126231e+11 -1.67127815e+13  8.74e+07 8.46e+00  5.75e+08     1s
   5   5.27807744e+11 -6.66550852e+12  2.91e+07 1.47e+00  1.85e+08     1s
   6   4.18942890e+11 -4.55761468e+12  1.71e+07 8.37e-01  1.07e+08     1s
   7   3.29801494e+11 -2.41961110e+12  9.98e+06 2.77e-01  5.34e+07     1s
   8   2.16594512e+11 -8.25731183e+11  5.26e+06 2.07e-07  1.96e+07     1s
   9   1.41100029e+11 -3.39922169e+11  2.24e+06 2.51e-08  7.35e+06     1s
  10   1.05879095e+11 -3.01451058e+11  1.57e+06 1.85e-08  5.43e+06     1s
  11   8.37439957e+10 -2.15870228e+11  1.14e+06 1.26e-08  3.72e+06     1s
  12   7.40605744e+10 -1.41404848e+11  9.79e+05 7.13e-09  2.57e+06     1s
  13   6.94487961e+10 -1.38861636e+11  8.47e+05 6.90e-09  2.38e+06     1s
  14   5.63007771e+10 -9.27090827e+10  6.67e+05 5.01e-09  1.62e+06     1s
  15   4.94075972e+10 -7.24803372e+10  5.57e+05 3.96e-09  1.28e+06     2s
  16   4.56678859e+10 -2.30783048e+10  4.37e+05 1.21e-08  7.39e+05     2s
  17   4.59910999e+10 -1.34982723e+09  1.97e+05 2.74e-09  4.53e+05     2s
  18   5.12149968e+10  4.45512894e+10  1.06e-03 4.94e-08  5.02e+04     2s
  19   4.94778360e+10  4.90044421e+10  1.46e-04 2.14e-08  3.57e+03     2s
  20   4.92784509e+10  4.92550684e+10  3.78e-06 3.46e-08  1.76e+02     2s
  21   4.92762167e+10  4.92688842e+10  1.10e-07 9.33e-08  5.52e+01     2s
  22   4.92758940e+10  4.92741439e+10  1.49e-08 6.66e-08  1.31e+01     2s
  23   4.92758428e+10  4.92748797e+10  7.96e-08 3.47e-07  7.22e+00     3s
  24   4.92758023e+10  4.92751144e+10  6.09e-08 2.39e-07  5.30e+00     3s
  25   4.92757487e+10  4.92754131e+10  4.27e-08 1.39e-07  2.59e+00     3s
  26   4.92757094e+10  4.92755147e+10  2.68e-08 1.61e-07  1.54e+00     4s
  27   4.92756830e+10  4.92755870e+10  2.12e-08 2.31e-07  8.54e-01     4s
  28   4.92756777e+10  4.92756219e+10  2.61e-08 2.68e-07  5.85e-01     4s
  29   4.92756687e+10  4.92756428e+10  2.98e-08 2.94e-07  3.81e-01     5s

Barrier solved model in 29 iterations and 4.58 seconds (1.29 work units)
Optimal objective 4.92756687e+10


INFO:linopy.constants: Optimization successful:
Status: ok
Termination condition: optimal
Solution: 78843 primals, 166469 duals
Objective: 4.93e+10
Solver model: available
Solver message: 2

INFO:pypsa.optimization.optimize:The shadow-prices of the constraints
StorageUnit-ext-p_dispatch-lower, StorageUnit-ext-p_dispatch-upper, StorageUnit-
fix-p_store-lower, StorageUnit-fix-p_store-upper, StorageUnit-ext-p_store-lower,
StorageUnit-ext-p_store-upper, StorageUnit-fix-state_of_charge-lower,
StorageUnit-fix-state_of_charge-upper, StorageUnit-ext-state_of_charge-lower,
StorageUnit-ext-state_of_charge-upper were not assigned to the network.
```

[39]:
```
PyPSA Network
Components:
 - Bus: 1
 - Carrier: 4
 - Generator: 2
 - Load: 1
 - StorageUnit: 2
Snapshots: 8760
```

The following 'build_base_network' repeats the commands in the above, i.e., build network, add generators and storage units, etc.:

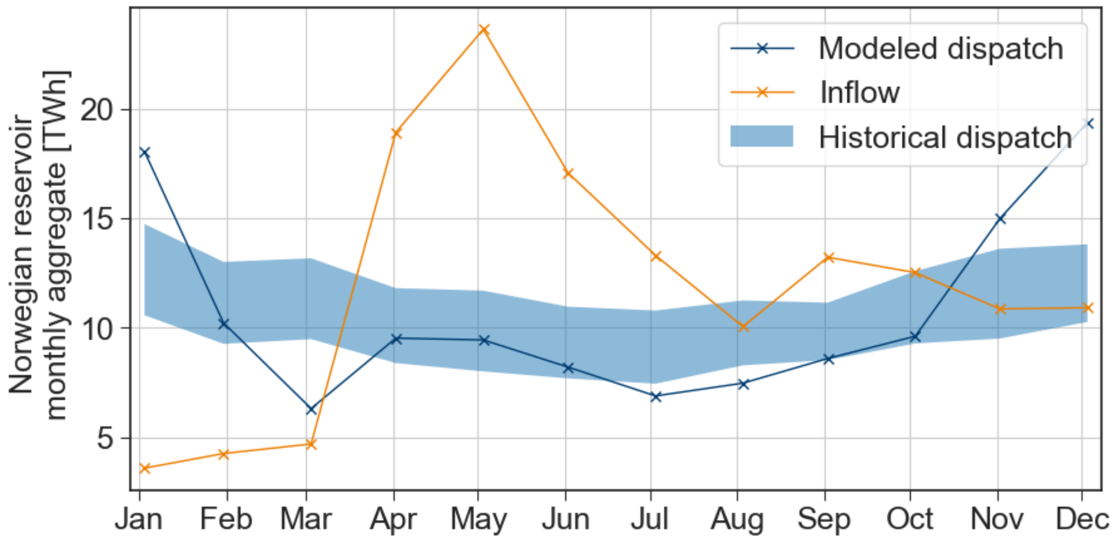
From this, we can find variables and constraint definitions.

[40]:
```
plot_hydro_operation(n_c,
                     df,
                     hydro_inflow,
                     freq='m'
                    )
```

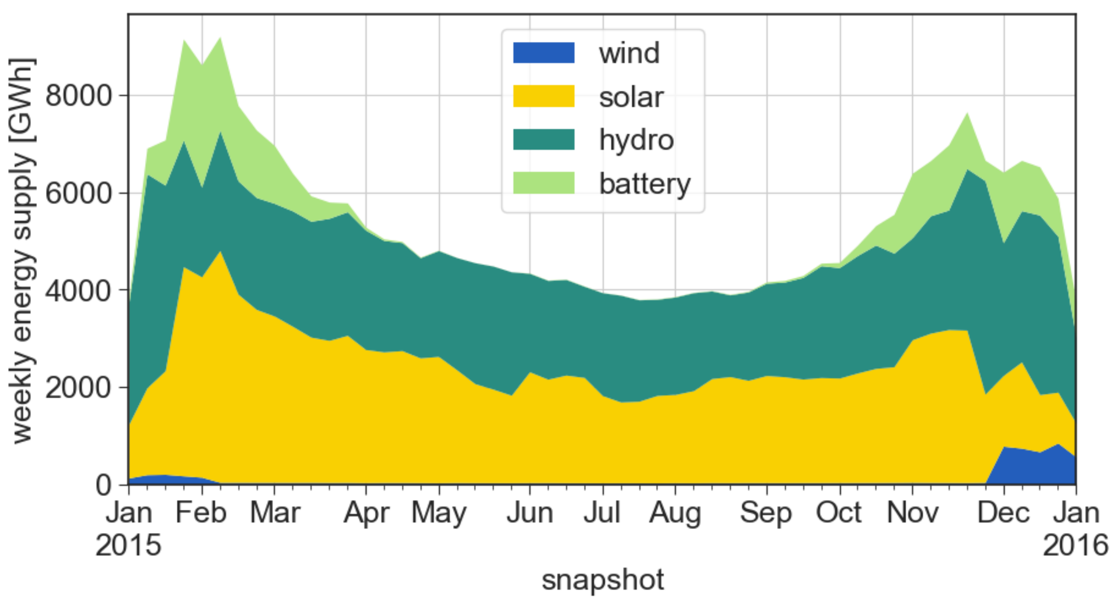


Comparison with the unconstrained scenario:

[41]:
```
plot_hydro_operation(n,
                     df,
                     hydro_inflow,
                     freq='m')
```

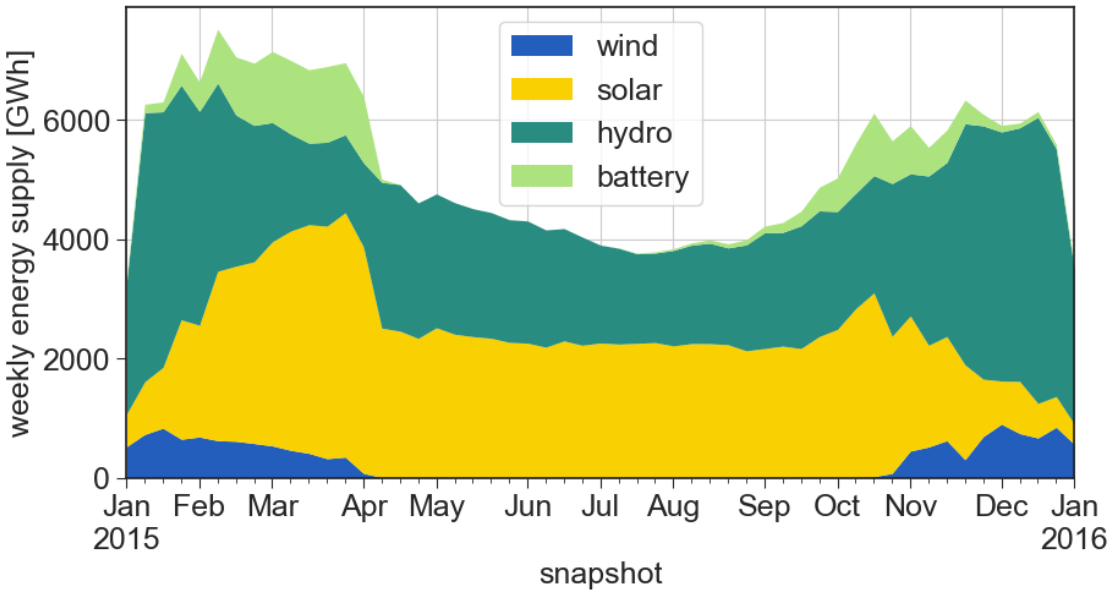


[42]:
```
plot_electricity_supply(n_c,
                        tech_colors,
                        freq='w')
```

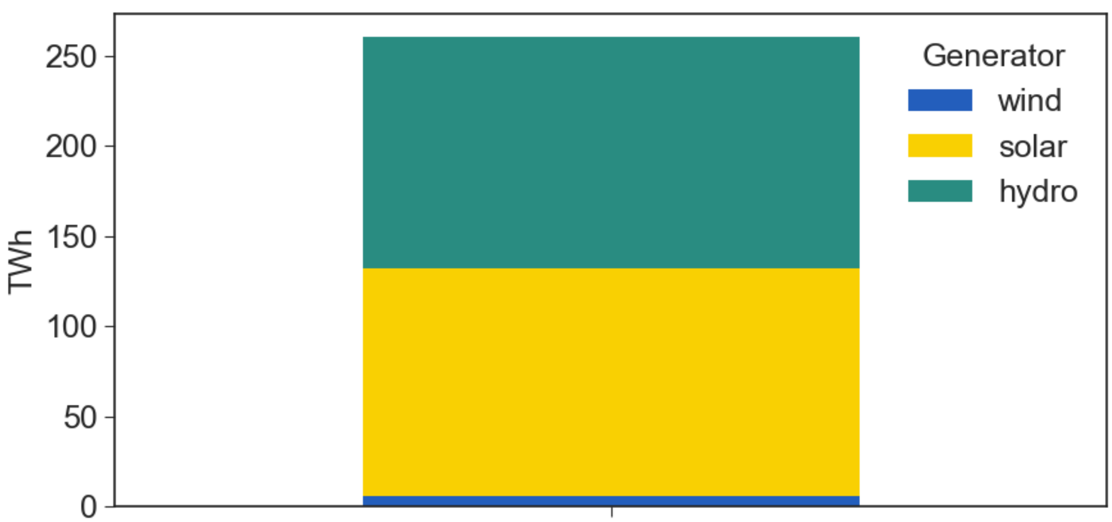


Comparison with unconstrained scenario:

[43]:
```
plot_electricity_supply(n,
                        tech_colors,
                        freq='w')
```

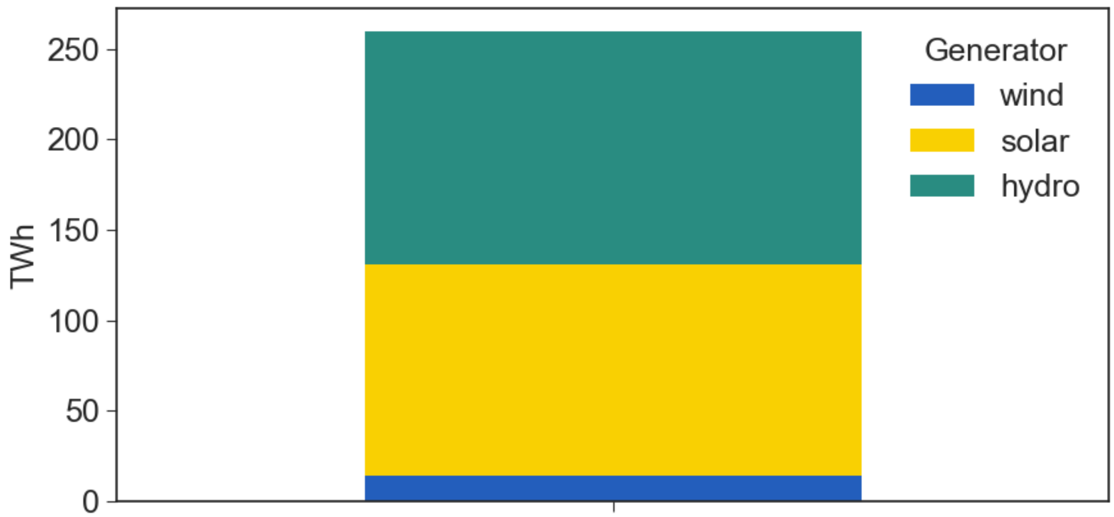


[44]:
```
plot_total_electricity_supply(n_c,
                              tech_colors)
```



Comparison with unconstrained scenario:

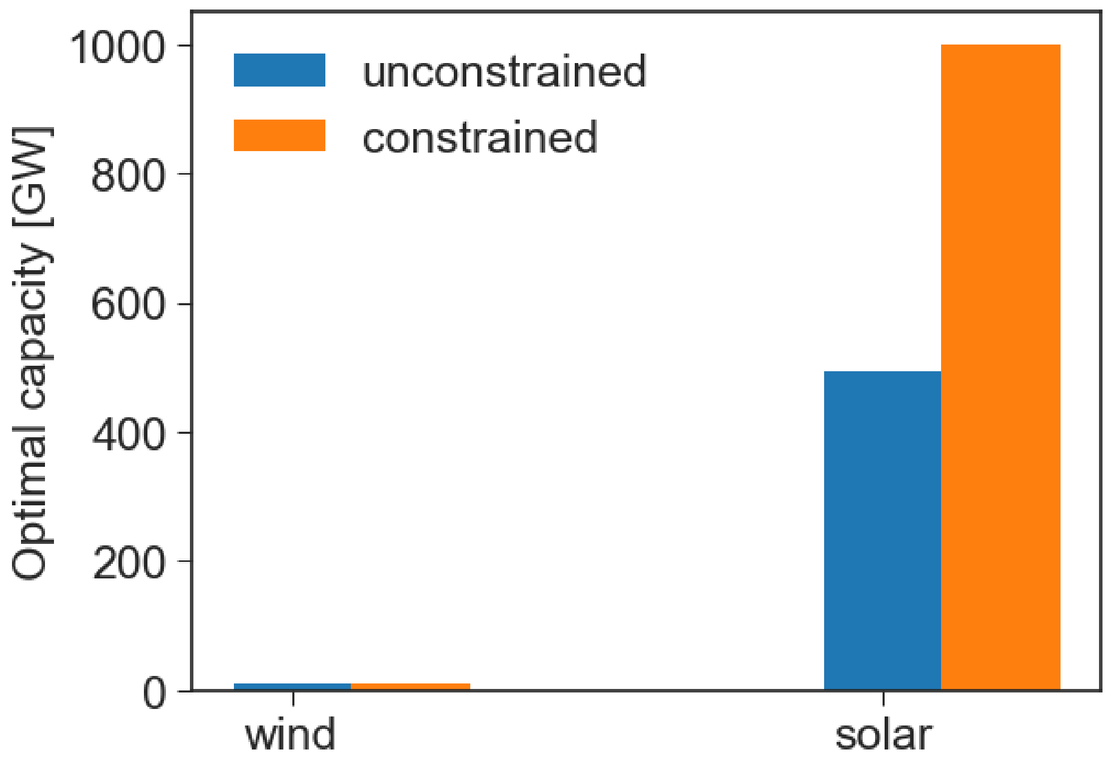
[45]:
```
plot_total_electricity_supply(n,
                              tech_colors)
```



Plot generation capacities:

[46]:
```
df = pd.DataFrame(n.generators.p_nom_opt)
generators = df.loc[pd.Index(['wind','solar'])]/1e3

df_c = pd.DataFrame(n_c.generators.p_nom_opt)
generators_c = df_c.loc[pd.Index(['wind','solar'])]/1e3

fig,ax = plt.subplots()
ax.bar([0,1],generators.p_nom_opt,width=0.2,label='unconstrained')
ax.bar([0.2,1.2],generators_c.p_nom_opt,width=0.2,label='constrained')
ax.set_xticks([0,1])
ax.set_xticklabels(generators.index)
ax.legend()
ax.set_ylabel('Optimal capacity [GW]')
```

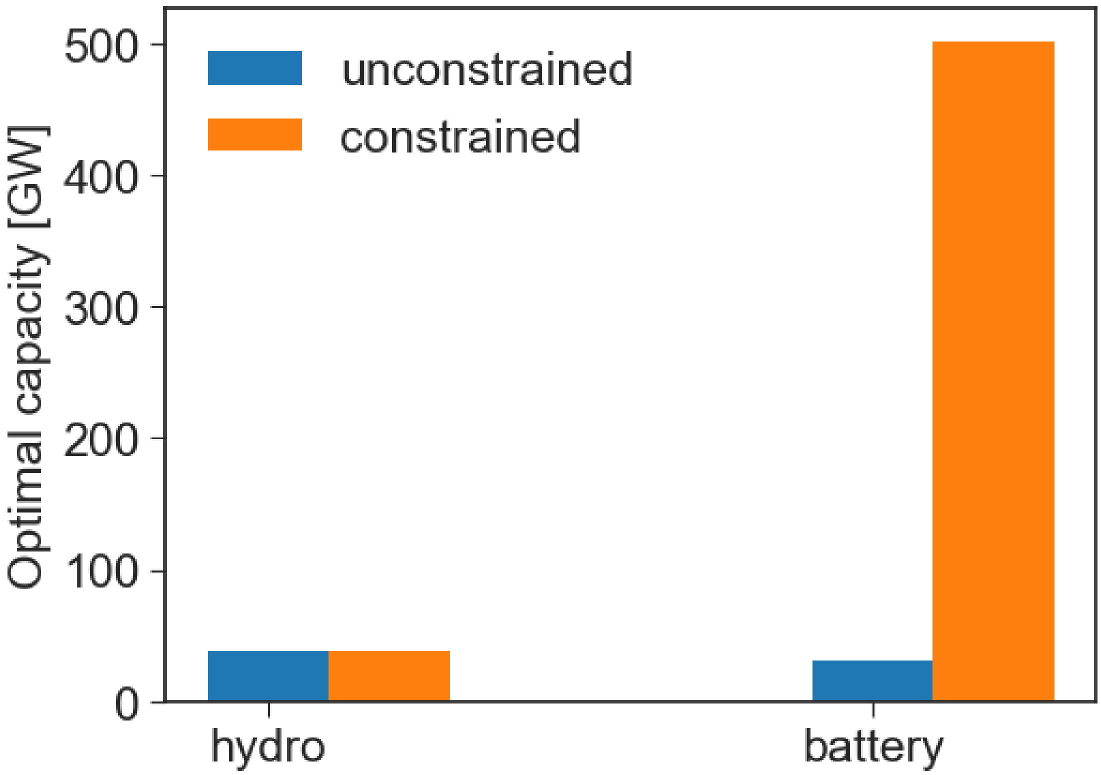
[46]:
```
Text(0, 0.5, 'Optimal capacity [GW]')
```



Plot storage capacities:

[47]:
```
df = pd.DataFrame(n.storage_units.p_nom_opt)
storage_units = df.loc[pd.Index(['hydro','battery'])]/1e3

df_c = pd.DataFrame(n_c.storage_units.p_nom_opt)
storage_units_c = df_c.loc[pd.Index(['hydro','battery'])]/1e3

fig,ax = plt.subplots()
ax.bar([0,1],storage_units.p_nom_opt,width=0.2,label='unconstrained')
ax.bar([0.2,1.2],storage_units_c.p_nom_opt,width=0.2,label='constrained')
ax.set_xticks([0,1])
ax.set_xticklabels(storage_units.index)
ax.legend()

ax.set_ylabel('Optimal capacity [GW]')
```

[47]:
```
Text(0, 0.5, 'Optimal capacity [GW]')
```


unconstrained
constrained
Optimal capacity [GW]
500
400
300
200
100
0
hydro
battery