# Prototype JulES v1

Check out the ReadMe page on Github for a description of JulES. This is a demo of the prototype implementation of JulES in a stepwise manner. Here are some of the most important steps:

- Load the TuLiPa and JulES code on parallel processors so that we can run scenarios and subsystems in parallel
- Decide on simulation and scenario parameters
- First time step in the simulation and initializing
  - Load datasets for the European power system/market.
  - Initialize and run price prognosis models
  - Do scenario modelling for the stochastic subsystem models
  - Initialize and run the stochastic subsystem models
  - Initialize and run the market clearing problem
  - Collect results
- Simulate the next time steps
  - Run price prognosis models, scenario modelling, stochastic subsystem models, market clearing and collect results for each time step
- Postprocess, store and plot results

## Import packages

In [1]:
```
using DataFrames, Plots, Statistics, JSON, Distributed
plotlyjs();
```

**The WebIO Jupyter extension was not detected. See the WebIO Jupyter integration documentation for more information.**

## Prepare parallell processing - import code on all cores

The problem is simulated on 31 2.20 GHz processors running in parallel. TODO: Test on faster processors.

In [2]:
```
const numcores = 31

if nprocs() < numcores
    addprocs(numcores - nprocs())
end

@show nprocs();
```

```
nprocs() = 31
```

In [3]:
```
@everywhere include(joinpath(dirname(pwd()),"jgrc/TuLiPa/src/TuLiPa.jl"));
```

In [4]:
```
@everywhere include(joinpath(dirname(pwd()),"jgrc/JulES/src/JulES.jl"));
```

## Starting point and scenario settings

- Series simulation 30 years with moving horizons and two day time steps. Starting from model year 2025 and weather scenario 1981.
- 30 possible weather scenarios (1981-2010) that can be used to consider uncertainty (we use scenario modelling later)
- Scenarios are phased in after two days
  - First two days have perfect information from simulation/main scenario
  - Next 5 weeks combines simulation/main scenario and scenario X. Smooth transition.
  - After 5 weeks the simulation/main scenario is fully phased out.

In [5]:
```
datayear = 2025
scenarioyearstart = 1981
scenarioyearstop = 2011
totalscen = 30 # scenarios to consider uncertainty for

# Standard time for market clearing - perfect information so simple time type
tnormal = FixedDataTwoTime(getisoyearstart(datayear),getisoyearstart(scenarioyears

# Phasein settings
phaseinoffsetdays = 2 # also simulation step length
phaseinoffset = Millisecond(Day(phaseinoffsetdays)) # phase in straight away from
phaseindelta = Millisecond(Week(5)) # Phase in the second stage scenario over 5 we
phaseinsteps = 5 # Phase in second stage scenario in 5 steps

# Make scenario times for all uncertainty scenarios. List of tuples with tnormal,
totalscentimes = []
for scen in 1:totalscen
    scentnormal = FixedDataTwoTime(getisoyearstart(datayear),getisoyearstart(scenar
    scentphasein = PhaseinTwoTime(getisoyearstart(datayear),getisoyearstart(scenar

    push!(totalscentimes, (scentnormal, scentphasein, scen))
end
```

## Price prognosis models uses NVEs Europe dataset for the TheMA model

This is a simplified version of the dataset NVE uses for its long-term power market analyses (for example https://www.nve.no/energi/analyser-og-statistikk/langsiktig-kraftmarkedsanalyse/). The dataset consist of:

- Detailed thermal units, aggregated hydro (Res, RoR, PHS), inelastic wind and solar, demands (inelastic with cutoff / load shedding price), storages (hydro and battery), transmission (ATC) and price areas (endogenous and exogenous).
- Levels in 2021, 2025, 2030, 2040 and 2050 (e.g. commodity price levels, installed production/transmission/storage capacities etc.)
- Profiles (e.g. availability profiles for transmission or production units, commodity price profiles, weather scenario profiles 1981-2010 for demand, solar, wind and inflow etc.)

We cannot publish the full dataset, but many of the profiles for wind, solar, hydro and demand can be downloaded from https://www.nve.no/energi/analyser-og-statistikk/vaerdatasett-for-kraftsystemmodellene/.

### NB!

- In this demo we use the model year 2025. This scenario was made a couple of years ago and results should be considered outdated. A lot has happened in the power market since then. In addition the dataset is simplified.

In [6]:
```
sti_themadata = "data_fra_thema"

themastructure = json_to_elements(sti_themadata, "dataset_thema.json")
themaseries = json_to_elements(sti_themadata, "tidsserier_thema.json")

elements = vcat(themaseries,themastructure)
addscenariotimeperiod!(elements, scenarioyearstart, scenarioyearstop);
```

## Initialize and run price prognosis models for all scenarios

- Three levels of details: Long, medium and short term problems (run sequentially for each scenario)
- All 30 scenarios (run in parallell)
- Inputs:
  - Power market representation / Europe dataset. (And how to aggregate it)
  - Problem length and time resolution
    - Long: 5 years long horizon with 6-weekly resolution for hydro, divided into 4 dynamic load blocks per 6 weeks for power. The dynamic load blocks groups together hours with similar residual load, which gives fewer periods but more variation between the periods. The alternative would be to for example have daily resolution for power, which ignores the price variation during the day.
    - Med: 54 week long horizon with weekly resolution for hydro, divided into 4 dynamic load blocks per week for power
    - Short: Week long horizon with daily resolution for hydro, and two-hourly for power
- Outputs:
  - Long, medium and short term prices for use in stochastic subsystem models.
  - Storage values (water values) from medium for use as end condition in stochastic subsystem models.
  - Thermal end states from short for use as end condition in market clearing model

In [7]:
```
# Set horizons for price prognosis models
# Long
longhorizonduration = Millisecond(Week(5*52))
longhydroperiodduration = Millisecond(Week(6))

longrhsdata = StaticRHSAHData("Power", datayear, scenarioyearstart, scenarioyearst
longmethod = KMeansAHMethod()
longclusters = 4
longunitduration = Millisecond(Hour(6))

# Medium
```

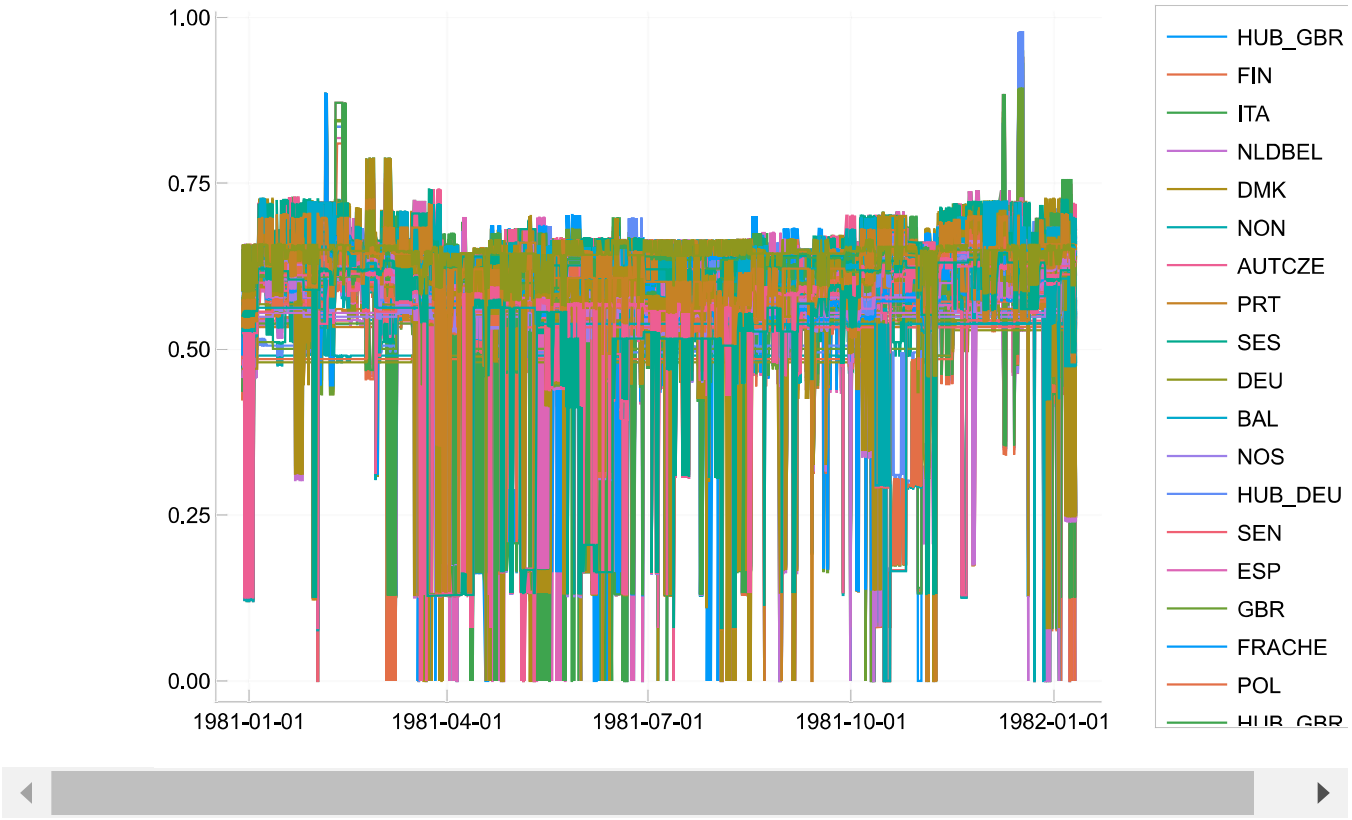
```
medhorizonduration = Millisecond(Week(54))
medhydroperiodduration = Millisecond(Day(7)); @assert medhorizonduration.value % l

medrhsdata = DynamicRHSAHData("Power")
medmethod = KMeansAHMethod()
medclusters = 4
medunitduration = Millisecond(Hour(4))

# Short
shorthorizonduration = Millisecond(Week(1))
shorthydroperiodduration = Millisecond(Day(1)); @assert medhorizonduration.value %
shortpowerparts = 12

# Preallocate storage for problems and results on different cores. Use package Dist
# Distribute scenarios
allscenarios = distribute(totalscentimes)

# Problems are built, updated, solved, and stored on a specific core. Moving a prob
longprobs = distribute([HiGHS_Prob() for i in 1:length(allscenarios)], allscenario
medprobs = distribute([HiGHS_Prob() for i in 1:length(allscenarios)], allscenarios
shortprobs = distribute([HiGHS_Prob() for i in 1:length(allscenarios)], allscenari

# Results are moved between cores. These are much smaller than longprobs/medprobs/s
medprices = distribute([Dict() for i in 1:length(allscenarios)], allscenarios)
shortprices = distribute([Dict() for i in 1:length(allscenarios)], allscenarios)
medendvaluesobjs = distribute([EndValues() for i in 1:length(allscenarios)], allsc
nonstoragestates = distribute([Dict{StateVariableInfo, Float64}() for i in 1:lengt

# Organise inputs and outputs
probs = (longprobs, medprobs, shortprobs)
longinput = (longhorizonduration, longhydroperiodduration, longrhsdata, longmethod
medinput = (medhorizonduration, medhydroperiodduration, medrhsdata, medmethod, med
shortinput = (shorthorizonduration, shorthydroperiodduration, shortpowerparts, "sho
allinput = (numcores, elements, allscenarios, phaseinoffset)
output = (medprices, shortprices, medendvaluesobjs, nonstoragestates)

# Initialize price prognosis models and run for first time step. Run scenarios in p
@time pl_prognosis_init!(probs, allinput, longinput, medinput, shortinput, output)
```

```
 86.040697 seconds (5.53 M allocations: 252.401 MiB, 1.22% compilation time)
```

Out[7]:
```
Task (done) @0x00000000744e3650
```

## Plot medium term prices for some scenarios

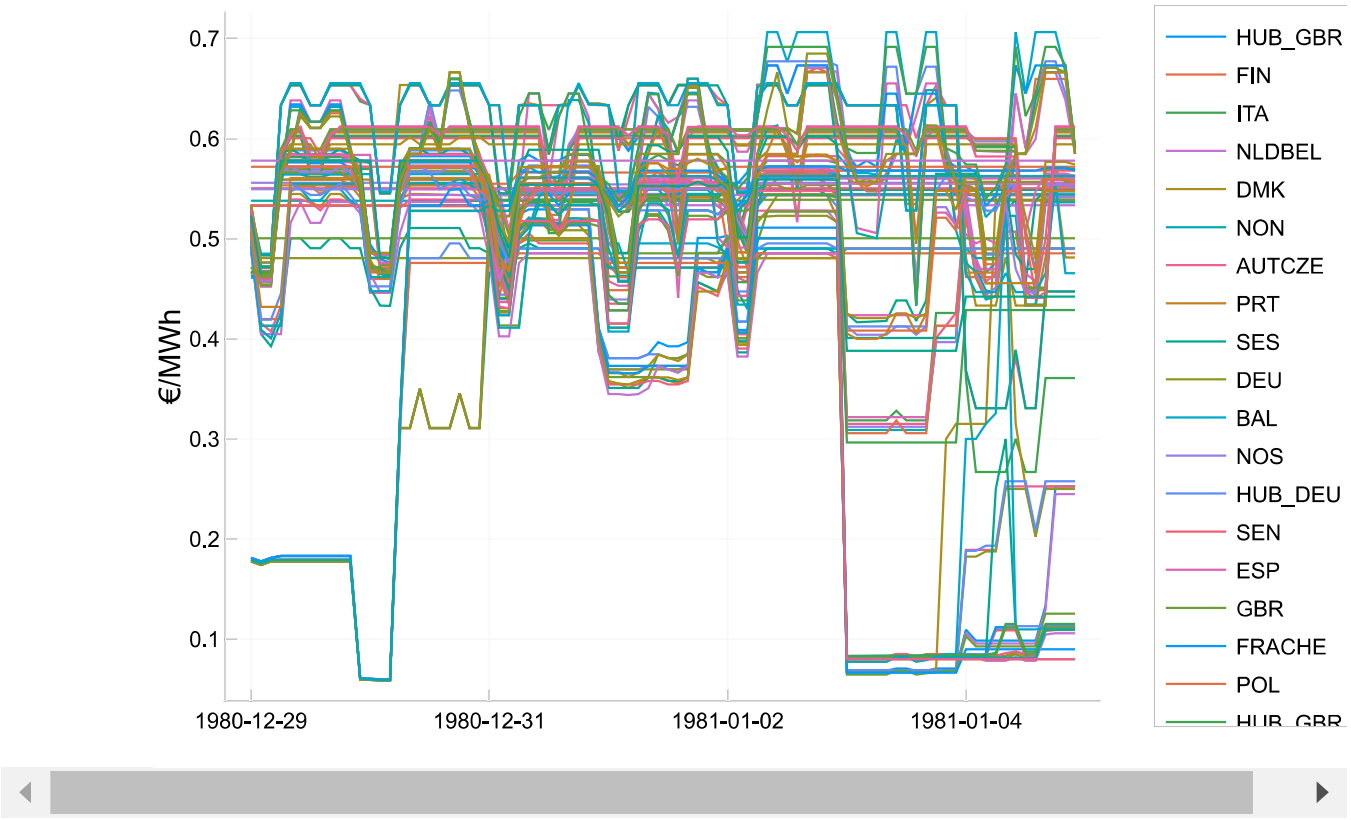
In [8]:
```
scenario = 1
index = collect(medprices[scenario]["steprange"])
prices = medprices[scenario]["matrix"]
labels = [name for name in medprices[scenario]["names"]]
p = plot(index,prices,label=reshape(labels, 1, length(labels)),legend=:outertoprigh

for scenario in 2:5 # length(allscenarios)
    prices = medprices[scenario]["matrix"]
    labels = [name for name in medprices[scenario]["names"]]
    plot!(p,index,prices,label=reshape(labels, 1, length(labels)),legend=:outertopr
end
display(p)
```




## Plot short term prices for some scenarios

In [9]:

```
scenario = 1
index = collect(shortprices[scenario]["steprange"])
prices = shortprices[scenario]["matrix"]
labels = [name for name in shortprices[scenario]["names"]]
p = plot(index,prices,label=reshape(labels, 1, length(labels)), ylabel="€/MWh",leg

for scenario in 2:5 # length(allscenarios)
    prices = shortprices[scenario]["matrix"]
    labels = [name for name in shortprices[scenario]["names"]]
    plot!(p, index,prices ,label=reshape(labels, 1, length(labels)),legend=:outert
end
display(p)
```




## Stochastic subsystem models and market clearing model uses more detailed hydropower data

NVEs Europe dataset (explained above) combined with detailed hydropower for Norway and Sweden. The full detailed dataset contains for Nortwestern Europe:

- 32 price areas (9 exogen)
- 73 transmission lines (19 with ramping)
- 162 demands
- 88 batteries (normal, and representing demand response and V2G)
- 294 thermal plants (228 with start up costs)
- 100 wind and solar plants (aggregated)
- 1482 hydropower modules
  - 965 with production
  - 43 with pumps (includes PHS)
  - 998 with reservoirs
  - 788 restrictions (environmental, reservoir curves and ramping)
  - 90 PQ-curves (mostly Sweden)
  - Metadata for head dependency (nominal head, outlet level and reservoir curves) for some plants and pumps

NVEs dataset for the hydropower system in 2022 is open, but we have not published datasets for 2025/2030/2040/2050 since it would reveal investment plans. The dataset exist in several formats:

- Aggregated (for Res and RoR) production capacity, reservoir capacity, inflow and inflow profiles per price areas from https://www.nve.no/energi/analyser-og-statistikk/vaerdatasett-for-kraftsystemmodellene/

- Detailed watercourse descriptions from https://www.nve.no/energi/energisystem/vannkraft/modell-av-det-norske-vannkraftsystemet/. The dataset exist in two DETD-formats (per EMPS area (also includes rest modules for small-scale hydro) or per watercourse), and simplified in an Excel-format. The dataset used in this demo is derived from the excel-format with some differences:
  - Every water balance has its own module (i.e. a module with both regulated and unregulated inflow is split into two modules).
  - It has pq-kurves, reservoir curves (for head dependency) and environmental restrictions
  - It has ramping restrictions for hydropower release (only used in market clearing).
  - It has metadata for head dependency (nominal head, outlet level and reservoir curves) for some plants and pumps
- The inflow series for the detailed dataset can be found at https://www.nve.no/vann-og-vassdrag/hydrologiske-data/historiske-data/historiske-vannfoeringsdata-til-produksjonsplanlegging/.

In [10]:
```
# Read dataset for stochastic and market clearing models
sti_themadata = "data_fra_thema"
tst = JSON.parsefile(joinpath(sti_themadata, "dataset_thema_excl_hydro_nose.json")
themastructure = getelements(tst)
tse = JSON.parsefile(joinpath(sti_themadata, "tidsserier_thema.json"))
themaseries = getelements(tse, sti_themadata)

dse = JSON.parsefile("data_fra_dynmodell/tidsserier_detd.json")
detdseries = getelements(dse);
dst = JSON.parsefile("data_fra_dynmodell/dataset_detd.json")
detdstructure = getelements(dst);

detailedelements = vcat(themaseries,themastructure,detdseries,detdstructure)
addscenariotimeperiod!(detailedelements, scenarioyearstart, scenarioyearstop);
```

## Get water values for detailed hydro (DETD) from aggregated hydro (TheMA)

- With different representations of watercourses in different problems, we need a mapping
- TODO: Only use the detailed hydropower dataset and aggregate it for the price prognosis models. Now this aggregation is done outside of the model which gives the user less flexibility.

In [11]:
```
# Mapping between aggregated and detailed storages
detailedrescopl = JSON.parsefile("data_fra_dynmodell/magasin_elspot1.json")

# Global energy equivalent detailed reservoirs
enekvglobaldict = Dict()
for element in detailedelements
    if element.typename == GLOBALENEQKEY
        enekvglobaldict[split(element.instancename,"GlobalEneq_")[2]] = element.va
    end
end

# Detailed dataset has reservoirs for SE4, aggregated does not, TODO: Improve aggre
for k in keys(detailedrescopl)
    if detailedrescopl[k] == "SE4"
```

```
        detailedrescopl[k] = "SE3"
    end
end

# Get dictionary with each detailed reservoir and their water value for each scenar
# TODO: Detailed run-of-river reservoirs get water value from aggregated reservoir
function getendvaluesdicts(endvaluesobjs, detailedrescopl::Dict, enekvglobaldict::D
    endvaluesdicts = Dict[]
    for endvaluesobj in endvaluesobjs
        instance = [getinstancename(getid(obj)) for obj in endvaluesobj.objects]
        endvalues = endvaluesobj.values
        endvaluesdict = Dict(instance .=> endvalues)

        for (k,v) in detailedrescopl
            endvaluesdict["Reservoir_" * k] = endvaluesdict["Reservoir_" * v * "_hy
        end
        push!(endvaluesdicts, endvaluesdict)
    end

    return endvaluesdicts
end
medendvaluesdicts = getendvaluesdicts(medendvaluesobjs, detailedrescopl, enekvgloba
```

## Scenario modelling

We use scenario modelling to consider uncertainty from 30 scenarios with only 7 scenarios. Scenarios can be chosen and weighted with different methods.

- In this prototype we use InflowClusteringMethod which cluster together scenarios with similar total energy inflows in the whole dataset (both level and profile). One scenario from each cluster will represent the others with the weight based on the size of the cluster.

This is implemented with modularity in mind, so you can chose other methods aswell or implement your own:

- NoScenarioModellingMethod keeps all the 30 scenarios
- SumInflowQuantileMethod calculates the total inflow in each scenario, puts them on a Bell curve and lets the user choose scenarios based on quantiles of the distribution. The user can also decide how to weight the scenarios with a polynomial function (for example weight the extremes more, only the wet years or only the dry years more)

TODO:

- Now we do the scenario modelling based on the energy inflow in the whole dataset. This works ok since the most important subsystem models are the watercourses in the Nordics. But the scenarios chosen will work worse for the battery systems. In the future we would like to have different scenario modelling for different technologies and systems in different geographical areas. Other alternatives could be to do the scenario modelling based on the price series, or with the residual load (including energy inflow.)
- We also want to implement more complex scenario generation. This modular design of JulES gives us the flexibility to in the future generate inflow series for each watercourse with state information like snow reservoir levels and weather forecasts.

In [12]:
```
# Scenario reduction to this amount
numscen = 7

# Modelobjects that can be used to reduce scenarios
scenarioelements = copy(detailedelements)

# Horizons are needed to build modelobjects, but not used in scenario modelling
dummyperiods = 10
dummyperiodduration = Millisecond(Hour(24))
power_horizon = SequentialHorizon(dummyperiods, dummyperiodduration)
hydro_horizon = SequentialHorizon(dummyperiods, dummyperiodduration)

set_horizon!(scenarioelements, "Power", power_horizon)
set_horizon!(scenarioelements, "Battery", power_horizon)
set_horizon!(scenarioelements, "Hydro", hydro_horizon)

scenarioobjects = collect(values(getmodelobjects(scenarioelements)))

# Scenario modelling method
scendelta = MsTimeDelta(Day(364)) # scenario modelling based on the next year, ever
parts = 4 # divide scendelta into this many parts, calculate sum inflow for each pa
scenmodmethod = InflowClusteringMethod(numscen, parts)
# scenmodmethod = NoScenarioModellingMethod(numscen, totalscentimes)
@time scenariomodelling!(scenmodmethod, scenarioobjects, numscen, totalscentimes, s
```

```
  5.455343 seconds (15.31 M allocations: 699.816 MiB, 3.38% gc time, 96.53% compil
ation time)
```

## Stochastic subsystem models with Benders decomposition

- Each unique storage system (watercourses or batteries) is solved as a two-stage stochastic LP with Benders decomposition
- The subsystems are optimized against exogen prices from the price prognosis scenarios
- We group storage systems into two. These are solved with different degree of detail
  - Med-term - mostly hydro
    - 52 week long horizon, weekly resolution, price from medium term price prognosis
  - Short-term - mostly batteries and PHS
    - Week long horizon, two-hourly resolution, price from short term price prognosis
- Outputs:
  - Cuts (battery or hydro) for use as end condition in market clearing problem

In [13]:
```
# Cut parameters
maxcuts = 13 # preallocate fixed number of cuts, no cut selection
lb = -1e5 # lower bound of the future value in the first iteration
reltol = 0.0001 # relative tolerance

# Parameters for stochastic subsystem problems (could also split totalduration into
shortstartstorage = 50
medstartstorage = 65
smpdp = Millisecond(Hour(2)) # short/med - master/sub - period duration - power/hyd
smpdh = Millisecond(Hour(2))
sspdp = Millisecond(Hour(2))
sspdh = Millisecond(Hour(2)) # both master and subproblems for PHS and batteries ha
mmpdp = Millisecond(Hour(24))
mmpdh = Millisecond(Hour(24)) # daily resolution in hydro master problems
mspdp = Millisecond(Hour(168))
```

```
mspdh = Millisecond(Hour(168)) # weekly resolution in hydro subproblems
shorttotalduration = shorthorizonduration # total duration of master and subproble
medtotalduration = medhorizonduration - Millisecond(Week(2)) # we reuse prices for

# Make sure time resolution of hydro and power are compatible (TODO: Could add func
@assert ceil(Int64, phaseinoffset/smpdp) == ceil(Int64, phaseinoffset/smpdh)
@assert ceil(Int64, (shorttotalduration-phaseinoffset)/sspdp) == ceil(Int64, (short
@assert ceil(Int64, phaseinoffset/mmpdp) == ceil(Int64, phaseinoffset/mmpdh)
@assert ceil(Int64, (medtotalduration-phaseinoffset)/mspdp) == ceil(Int64, (medtota

# Convert DistributedArray of prices to local process
medpriceslocal = convert(Vector{Dict}, medprices)
shortpriceslocal = convert(Vector{Dict}, shortprices)

# Inputs
storageinfo = (shortstartstorage, medstartstorage, medendvaluesdicts)
shortterminputs = (detailedelements, shorttotalduration, smpdp, smpdh, sspdp, sspdl
medterminputs = (detailedelements, medtotalduration, mmpdp, mmpdh, mspdp, mspdh, s

ustoragesystemobjects = []
ushorts = []
# Make modelobjects for short-term subsystems
@time makemastersubobjects!(shortterminputs, ustoragesystemobjects, ushorts)
# Make modelobjects for medium-term subsystems
@time makemastersubobjects!(medterminputs, ustoragesystemobjects, ushorts)

# Distribute subsystems with inputs and outputs on different cores
storagesystemobjects, shorts = distribute_subsystems(ustoragesystemobjects, ushorts
masters = distribute([HiGHS_Prob() for i in 1:length(storagesystemobjects)], storag
subs = distribute([[] for i in 1:length(storagesystemobjects)], storagesystemobject
states = distribute([Dict{StateVariableInfo, Float64}() for i in 1:length(storages
cuts = distribute([SimpleSingleCuts() for i in 1:length(storagesystemobjects)], st
storagesystems = distribute([Dict() for i in 1:length(storagesystemobjects)], stora

# Initialize subsystem problems and run for first time step. Run subsystems in para
@time pl_stochastic_init!(numcores, storagesystemobjects, shorts, masters, subs, st
```

```
  5.879978 seconds (16.36 M allocations: 706.884 MiB, 2.90% gc time, 52.02% compil
ation time)
  3.243251 seconds (12.81 M allocations: 518.070 MiB, 4.68% gc time, 2.44% compila
tion time)
  6.915521 seconds (2.88 M allocations: 100.548 MiB, 4.53% compilation time)
```

Out[13]:
```
Task (done) @0x00000000735f22f0
```

## Market clearing problem

- Deterministic LP
- Two days, full detail (hydro and thermal), two-hourly resolution for power, 6 hourly resolution for hydro
- Inputs:
  - Detailed power market representation
  - Water values and battery storage values from stochastic subsystem problems
  - Thermal end condition from short term price prognosis model
- Outputs:
  - Two day detailed power market simulation (price, production, reservoir etc...)
  - Start state for next time step (reservoirs and other state variables)

In [14]:
```
# Bring data to local core
masterslocal = convert(Vector{Prob}, masters)
```

```
cutslocal = convert(Vector{SimpleSingleCuts}, cuts)
nonstoragestateslocal = convert(Vector{Dict}, nonstoragestates)

# Initialize market clearing problem and run for first time step
cpdp = Millisecond(Hour(2)) # clearing period duration power/battery
cnpp = ceil(Int64, phaseinoffset/cpdp) # clearing numperiods power/battery
cpdh = Millisecond(Hour(6)) # clearing period duration hydro
# cpdh = Millisecond(Hour(2)) # clearing period duration hydro
cnph = ceil(Int64, phaseinoffset/cpdh) # clearing numperiods hydro
@time clearing, nonstoragestatesmean, varendperiod = clearing_init(detailedelements
```

```
 12.962708 seconds (16.55 M allocations: 990.515 MiB, 5.99% gc time, 53.05% compil
ation time)
```

In [15]:
```
# Initialize start states for next time step, also mapping to TheMA storages and m
startstates = startstates_init(clearing, detailedrescopl, enekvglobaldict, longprol
```

### Head dependency in clearing_init!() and clearing!()

Inside the market clearing we have implemented head dependency:

- These functions sets state/head dependent energy equivalents for production and pumping
  - statedependentprod_init!(clearing, 65, t)
  - statedependentpump_init!(clearing, 65, t)
- We use the headloss cost method ReservoirCurveSlopeMethod which increases the watervalue if a small increase in reservoir filling would give a higher head at the current reservoir filling. The head loss cost is based on the slope of the reservoir curve (reservoir height [m] to filling [Mm3]) at the current reservoir filling. To get the headloss cost, we multiply the water value with a factor of the percentage change in the height for a 2% change in the reservoir filling divided by 2% (a bit simplified). An example of a reservoir with a steep reservoir curve is Storglomvatn belonging to Svartisen, so the headloss cost will incentivise a higher reservoir filling to get a higher head and energy equivalent at production.
  - updateheadlosscosts!(ReservoirCurveSlopeMethod(), clearing, masterslocal, t)

See the code at TuLiPa/reasoning_nvehydro.jl

TODO: Make an interface for head dependency to support different methods. The method used in this demo is relatively simple since we only consider head dependency in the market clearing, but we have methods that also consider head dependency in the master and subproblems of the stochastic subsystem models.

## Initialize and collect results: Prices and start states (e.g. reservoirs)

In [16]:
```
# Initialize and collect prices and start states
price = Dict()
powerhorizonix = argmax(getnumperiods(h) for h in clearing.horizons)
getareaprices!(price, clearing, clearing.horizons[powerhorizonix], tnormal)
areanames = price["names"]
ix = collect(price["steprange"])
pricematrix = copy(price["matrix"])
```

```
statenames = collect(keys(startstates))
statematrix = collect(values(startstates));
```

### Collect more detailed results

- Prices, supply, demand, transmission and hydro storage levels

In [17]:
```
clearingobjects = Dict(zip([getid(obj) for obj in clearing.objects],clearing.objec
# resultobjects = getpowerobjects(clearingobjects,["NO2"]); # only collect results
resultobjects = clearing.objects # collect results for all areas
prices, rhstermvalues, production, consumption, hydrolevels, batterylevels, powerb
```

## Simulate next time steps (16 steps = 32 days)

- Simulate next time steps, store results and plot results

In [18]:
```
# Only do scenario modelling and calculate new cuts every 8 days (other reuse scen
skipmed = Millisecond(Day(0))
skipmax = Millisecond(Day(6))

# How many time steps to run the simulation for
steps = 16;
# steps = (getisoyearstart(scenarioyearstop) - getisoyearstart(scenarioyearstart))
```

In [19]:
```
step = 2 # already ran first step in initialization

totaltime = @elapsed while step <= steps # while step <= steps and count elapsed t
    # Increment simulation/main scenario and uncertainty scenarios
    tnormal += phaseinoffset
    display(tnormal)

    for i in 1:length(totalscentimes)
        (scentnormal, scentphasein, scenario) = totalscentimes[i]
        scentnormal += phaseinoffset
        scentphasein = PhaseinTwoTime(getdatatime(scentnormal), getscenariotime(sc
        totalscentimes[i] = (scentnormal, scentphasein, scenario)
    end

    # Increment skipmed - should we reuse watervalues this time step?
    skipmed += Millisecond(phaseinoffset)
    if skipmed > skipmax
        skipmed = Millisecond(0)
    end

    # Deterministic long/mid/short - calculate scenarioprices for all 30 scenarios
    allscenarios = distribute(totalscentimes, allscenarios) # TODO: Find better so
    @time pl_prognosis!(numcores, longprobs, medprobs, shortprobs, medprices, shor

    # Stochastic sub systems - calculate storage value
    if skipmed.value == 0
        # Choose new scenarios
        scenariomodelling!(scenmodmethod, scenarioobjects, numscen, totalscentimes

        medpriceslocal = convert(Vector{Dict}, medprices)
        medendvaluesdicts = getendvaluesdicts(medendvaluesobjs, detailedrescopl, e
    else
        # Increment existing scenarios
        for i in 1:length(scenmodmethod.scentimes)
            (scentnormal, scentphasein, scenario) = scenmodmethod.scentimes[i]
            scentnormal += phaseinoffset
```

```
            scentphasein = PhaseinTwoTime(getdatatime(scentnormal), getscenariotime
            scenmodmethod.scentimes[i] = (scentnormal, scentphasein, scenario)
        end
    end
    shortpriceslocal = convert(Vector{Dict}, shortprices)

    @time pl_stochastic!(numcores, masters, subs, states, cuts, startstates, medpr

    # Market clearing
    masterslocal = convert(Vector{Prob}, masters)
    cutslocal = convert(Vector{SimpleSingleCuts}, cuts)
    nonstoragestateslocal = convert(Vector{Dict}, nonstoragestates)

    @time clearing!(clearing, tnormal, startstates, masterslocal, cutslocal, nonst

    # Results
    updateareaprices!(price, clearing, clearing.horizons[powerhorizonix], tnormal)
    ix = vcat(ix, collect(price["steprange"]))
    pricematrix = vcat(pricematrix, price["matrix"])
    statematrix = hcat(statematrix, collect(values(startstates)))

    prices, rhstermvalues, production, consumption, hydrolevels, batterylevels = u

    # Increment step
    step += 1
end

display(string("The simulation took: ", totaltime/60, " minutes"))
display(string("Time usage per timestep: ", totaltime/steps, " seconds"))
```

```
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1980-12-31T00:00:00"))
  5.064821 seconds (183.66 k allocations: 10.678 MiB, 1.05% compilation time)
  1.899553 seconds (2.27 M allocations: 70.734 MiB, 4.35% compilation time)
 16.189062 seconds (5.11 M allocations: 360.168 MiB, 1.54% compilation time)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-02T00:00:00"))
  6.098245 seconds (72.94 k allocations: 5.026 MiB)
  0.636664 seconds (2.02 M allocations: 57.079 MiB)
 18.583206 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-04T00:00:00"))
  4.668130 seconds (72.90 k allocations: 5.025 MiB)
  0.782235 seconds (2.02 M allocations: 57.076 MiB)
 15.299216 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-06T00:00:00"))
  7.346379 seconds (72.92 k allocations: 5.022 MiB)
  2.614726 seconds (2.02 M allocations: 57.071 MiB)
 17.687297 seconds (4.94 M allocations: 351.412 MiB, 1.26% gc time)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-08T00:00:00"))
  4.175312 seconds (72.90 k allocations: 5.022 MiB)
  0.634420 seconds (2.02 M allocations: 57.079 MiB)
  9.802127 seconds (4.94 M allocations: 351.412 MiB, 2.29% gc time)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-10T00:00:00"))
  4.945015 seconds (72.91 k allocations: 5.024 MiB)
  0.644773 seconds (2.02 M allocations: 57.079 MiB)
 15.145280 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-12T00:00:00"))
  4.376527 seconds (72.90 k allocations: 5.025 MiB)
  0.853126 seconds (2.02 M allocations: 57.340 MiB, 22.98% gc time)
 19.037005 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-14T00:00:00"))
  7.611186 seconds (72.93 k allocations: 5.026 MiB)
  1.961129 seconds (2.02 M allocations: 57.070 MiB)
 15.986845 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-16T00:00:00"))
```

```
  4.567726 seconds (72.91 k allocations: 5.025 MiB)
  0.643079 seconds (2.02 M allocations: 57.083 MiB)
 18.156059 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-18T00:00:00"))
  5.646076 seconds (72.92 k allocations: 5.024 MiB)
  0.639058 seconds (2.02 M allocations: 57.075 MiB)
 13.278711 seconds (4.94 M allocations: 351.412 MiB, 1.69% gc time)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-20T00:00:00"))
  5.996736 seconds (72.94 k allocations: 5.026 MiB)
  0.642769 seconds (2.02 M allocations: 57.079 MiB)
 12.078575 seconds (4.94 M allocations: 351.412 MiB, 1.59% gc time)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-22T00:00:00"))
  8.668308 seconds (72.95 k allocations: 5.026 MiB)
  2.067262 seconds (2.02 M allocations: 57.067 MiB)
 10.712538 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-24T00:00:00"))
  6.282259 seconds (72.95 k allocations: 5.026 MiB)
  0.834117 seconds (2.02 M allocations: 57.338 MiB, 22.19% gc time)
 16.243282 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-26T00:00:00"))
  5.758296 seconds (72.98 k allocations: 5.029 MiB)
  0.654093 seconds (2.02 M allocations: 57.079 MiB)
 16.223951 seconds (4.94 M allocations: 351.340 MiB)
FixedDataTwoTime(DateTime("2024-12-30T00:00:00"), DateTime("1981-01-28T00:00:00"))
  5.134657 seconds (72.91 k allocations: 5.024 MiB)
  0.645502 seconds (2.02 M allocations: 57.079 MiB)
 12.573497 seconds (4.94 M allocations: 351.412 MiB, 1.95% gc time)
"The simulation took: 5.743335204999999 minutes"
"Time usage per timestep: 21.53750701875 seconds"
```

## Results

In the first phase of JulES we have prioritized implementing functionality and test that they give the intended results, and compared results against other long-term models (TheMA and Samnett, which give a sufficient basis for comparison) on a big dataset. The price formation, reservoir operation and runtime of JulES seems promising, but there remains a lot of testing to see all the impacts of the concept implementation, model parameters, scenario modelling and head dependency on water values, prices, reservoir operation and runtime. The testing has been restricted by the runtime of the model, which we will have to improve to test different parts of the model more efficiently. In the next phase we want to test JulES as a short/medium-term paralell-simulation prognosis model. The results can then be compared to our other prognosis model (EMPS) and quickly be compared to the real market and historical data.

Throughout the testing we have achieved the wanted price volatility in the thermal dominated part of the dataset (Western Europe). On the other hand, the Nordics have had very flat prices due to too much flexibility in the hydropower modelling. We have therefore added hydropower production ramping restrictions in the market clearing in an attempt to reduce the flexibility of the run-of-river hydropower plants. This results in much more price volatiliy, but at a big computational cost. Ramping restrictions on transmission lines has the same effect.

Time usage with the current implementation in the demo and serial simulation of 30 weather years (~5500 two day long time steps):

- Same as demo: 2 hourly power resolution and 6 hourly hydro resolution in market clearing and hydro ramping restrictions
  - 46 hours total or 30 sec per time step or 105 sec per week
- 2 hourly hydro resolution in market clearing and hydro ramping restrictions (no other difference to demo)
  - 78 hours total or 51 sec per time step or 180 sec per week
- 24 hourly hydro resolution in market clearing and no hydro ramping restrictions (no other difference to demo)
  - 20 hours total or 13 sec per time step or 45 sec per week

This is promising considering the big dataset, and the list of possible optimization we have in mind:

- At the moment we solve the market clearing problem with the HiGHS serial Simplex method, which could be replaced by a method that uses parallelization. The HiGHS PAMI Simplex method is a good option that is faster, but there are also other open-source solvers that should be tested.
- It is interesting what these computational times would be with a commercial solver (we now use HiGHS), and with more and faster processor cores in parallel (now 30 2.2 GHz processor cores).
- We could clear the market for 24 hours at a time instead of 48 hours like now, which could reduce the computational time depending on if the market clearing has a higher runtime than the stochastic subsystem and price prognosis models (this is the case when we have a very detailed market clearing).
- We could try different configurations of ramping restrictions, and test if time delays in watercourses can achieve the same effects at a lower computation cost. Considering unavailability of hydropower or reserve market obligations, should also decrease the flexibility of the hydropower system. Detailed transmission system modelling should also be implemented in the future.
- We could run the model only for the Nordics, which would reduce the size of the dataset substantially and give results that are comparable to other models we use.

The price levels in the Nordics are higher than in other models. This is partly due to the high flexibility in the hydropower modelling, which gives stable high prices and not many zero-prices. Another reason is that the stochastic subsystem models could need some improvements, for example longer and more detailed horizons. This should give water values that gives better long term signals. More load shedding can also be a contributor to higher prices, but this can be prevented with scenario modelling and head dependencies.

We have also seen the effects of scenario modelling and head dependencies. Scenario modelling can be used to reduce the runtime and adjust the risk taking, which gives more realistic reservoir operation and avoid the extremes of flooding and load shedding. Head dependency can be used to get a more realistic reservoir operation and higher production, and also gives lower risk of load shedding. These have to be tested further.

We are also very happy with the modelling choice of modularity, using time-series datasets and using Julia. This has made TuLiPa and JulES very pleasant to work with, as they provide a great deal of flexibility in adding complex functionality without having to make extensive

changes to the existing code. Additionally, the models can be run with different methods and time resolutions without adaptations of the dataset. These design choices contribute to the model's suitability for further development and modeling the future power system when new modeling requirements arise.

However, the project's codebase needs to be professionalized with better structure to make the model more user-friendly, allowing not only developers to run the model. Unit testing is also important to ensure that the model functions as intended. So far, we have been working on the model concept alone, so it will be crucial to involve analysts who will use the models and developers outside of NVE who can contribute to further developing the concept.

## Postprocess detailed results

- Combine fixed contributions (e.g. wind, solar and demand) together with supply and demand variables
- Make time axis for price, supply/demand and reservoir levels

In [20]:
```
# Only keep rhsterms that have at least one value (TODO: Do the same for sypply and
rhstermtotals = dropdims(sum(rhstermvalues,dims=1),dims=1)
rhstermsupplyidx = []
rhstermdemandidx = []

for k in 1:length(rhsterms)
    if rhstermtotals[k] > 0
        push!(rhstermsupplyidx, k)
    elseif rhstermtotals[k] < 0
        push!(rhstermdemandidx, k)
    end
end

# Put rhsterms together with supplies and demands
rhstermsupplyvalues = rhstermvalues[:,rhstermsupplyidx]
rhstermdemandvalues = rhstermvalues[:,rhstermdemandidx]*-1

rhstermsupplynames = [getinstancename(rhsterm) for rhsterm in rhsterms[rhstermsupp
rhstermsupplybalancenames = [split(getinstancename(r), "PowerBalance_")[2] for r in
rhstermdemandnames = [getinstancename(rhsterm) for rhsterm in rhsterms[rhstermdema
rhstermdemandbalancenames = [split(getinstancename(r), "PowerBalance_")[2] for r in

supplynames = [[getinstancename(plant) for plant in plants];rhstermsupplynames]
supplybalancenames = [[split(getinstancename(p), "PowerBalance_")[2] for p in plan
supplyvalues = hcat(production,rhstermsupplyvalues)

demandnames = [[getinstancename(demand) for demand in demands];rhstermdemandnames]
demandbalancenames = [[split(getinstancename(p), "PowerBalance_")[2] for p in dema
demandvalues = hcat(consumption, rhstermdemandvalues)

# Prepare for plotting results
hydronames = [getinstancename(hydro) for hydro in hydrostorages]
batterynames = [getinstancename(battery) for battery in batterystorages]
powerbalancenames = [split(getinstancename(getid(powerbalance)), "PowerBalance_")[

# Convert reservoir filling to TWh
hydrolevels1 = copy(hydrolevels)
for (i,hydroname) in enumerate(hydronames)
    if haskey(getbalance(clearingobjects[hydrostorages[i]]).metadata, GLOBALENEQKE
        hydrolevels1[:,i] = hydrolevels1[:,i]*getbalance(clearingobjects[hydrostor
```

```
    end
end

# Time
x1 = [getisoyearstart(scenarioyearstart) + cpdp*(t-1) for t in 1:first(size(supply
x2 = [getisoyearstart(scenarioyearstart) + cpdh*(t-1) for t in 1:first(size(hydrol
```

In [21]:

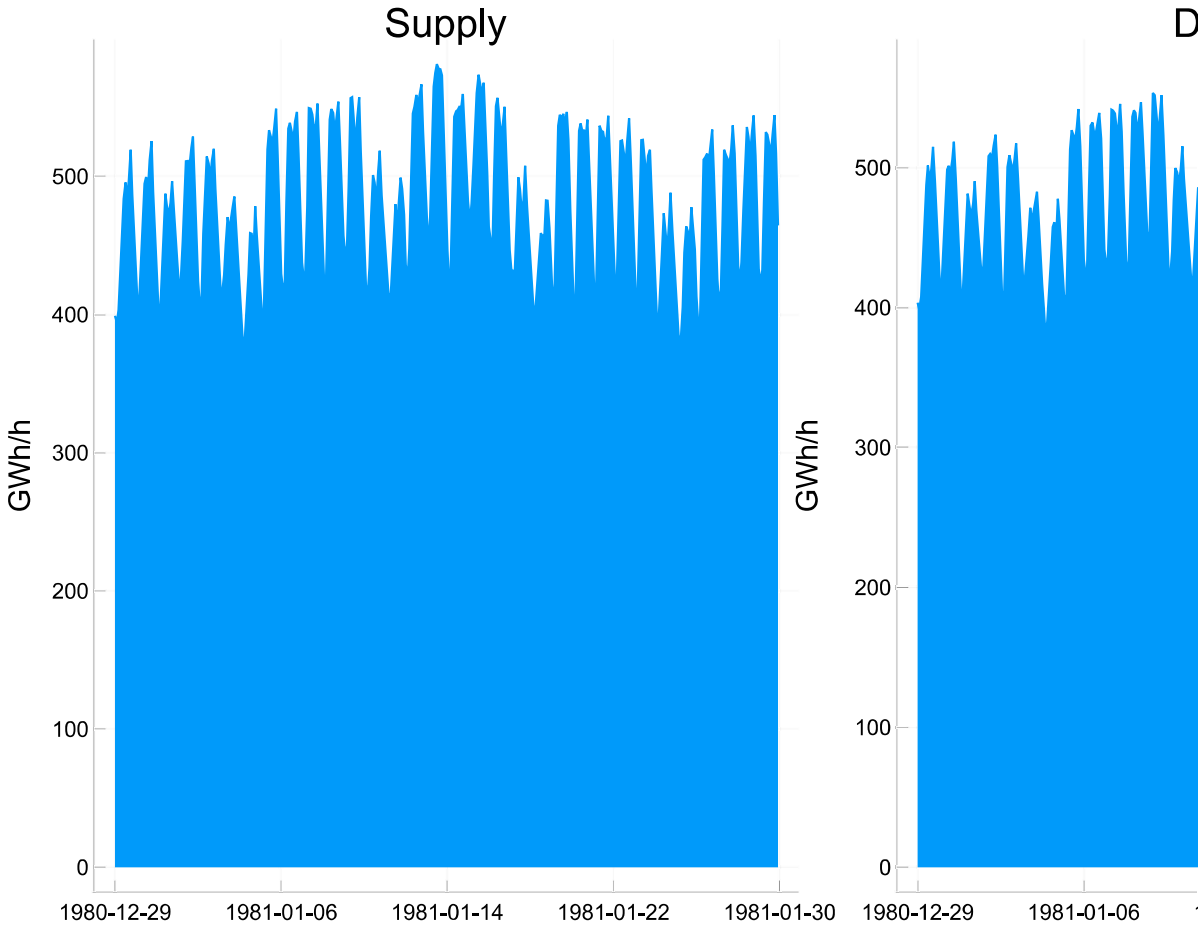
```
# # Store results with binary h5 format
# datetimeformat = "yyyy-mm-ddTHH:MM:SS"
# modelname = JulES_2030_81_10

# data = Dict()
# data["areanames"] = areanames |> Vector{String}
# data["pricematrix"] = pricematrix
# data["priceindex"] = Dates.format.(ix, datetimeformat) # not necessary to store

# data["statenames"] = statenames
# data["statematrix"] = permutedims(statematrix)
# data["stateindex"] =  Dates.format.([getisoyearstart(scenarioyearstart) + Day(2*

# data["supplyvalues"] = supplyvalues
# data["supplynames"] = supplynames
# data["supplybalancenames"] = supplybalancenames

# using FileIO, HDF5
# @time h5open("$modelname.h5", "w") do file
#     for (k,v) in data
#         display(k)
#         write(file, k, v)
#     end
# end

# # # Read results so that they can be compared to other models (we have another s
# # # Read using JLD2
# # using JLD2
# # @time JulESdata = JLD2.load("$modelname.h5")

# # # Read using HDF5
# # JulESdata = Dict{String, Any}()
# # @time c = h5open("$modelname.h5", "r") do file
# #     for key in keys(data)
# #         JulESdata[key] = read(file, key)
# #     end
# # end

# # Store as CSV
# areaprices = rename!(DataFrame(prices, :auto),powerbalancenames)
# areaprices[!,:time] = x1
# CSV.write("JulES_results\\prices.csv", areaprices)

# demand = rename!(DataFrame(demandvalues, :auto),demandnames)
# demand[!,:time] = x1
# demand = stack(demand,Not(:time))
# demandcopl = DataFrame(variable=demandnames, area=demandbalancenames)
# demand = leftjoin(demand, demandcopl, on=:variable)
# CSV.write("JulES_results\\demand.csv", demand)

# supply = rename!(DataFrame(supplyvalues, :auto),supplynames)
# supply[!,:time] = x1
# supply = stack(supply,Not(:time))
# supplycopl = DataFrame(variable=supplynames, area=supplybalancenames)
# supply = leftjoin(supply, supplycopl, on=:variable)
# CSV.write("JulES_results\\supply.csv", supply)
```

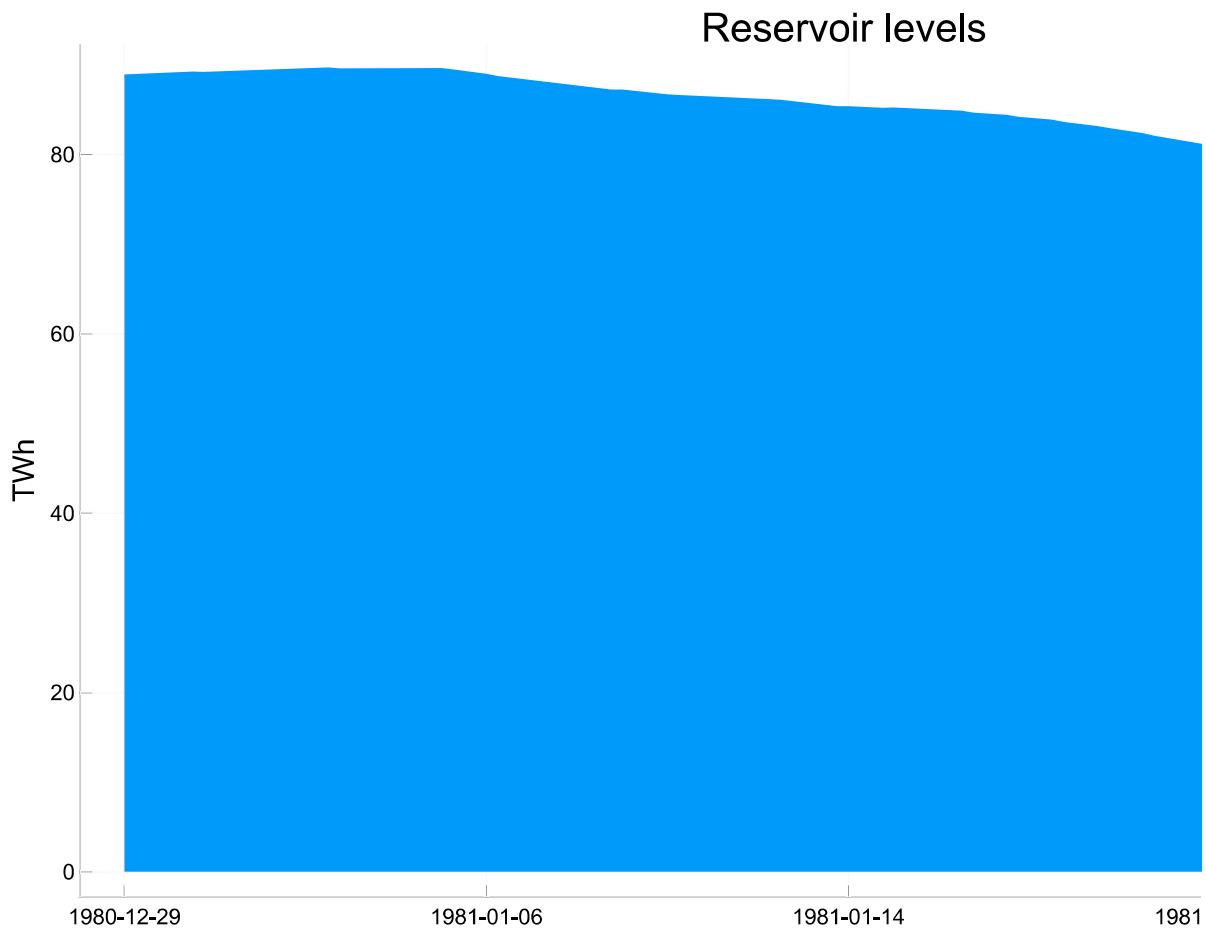
```
# hydro = rename!(DataFrame(hydrolevels, :auto),hydronames)
# hydro[!,:time] = x2
# CSV.write("JulES_results\\hydro.csv", hydro);
```

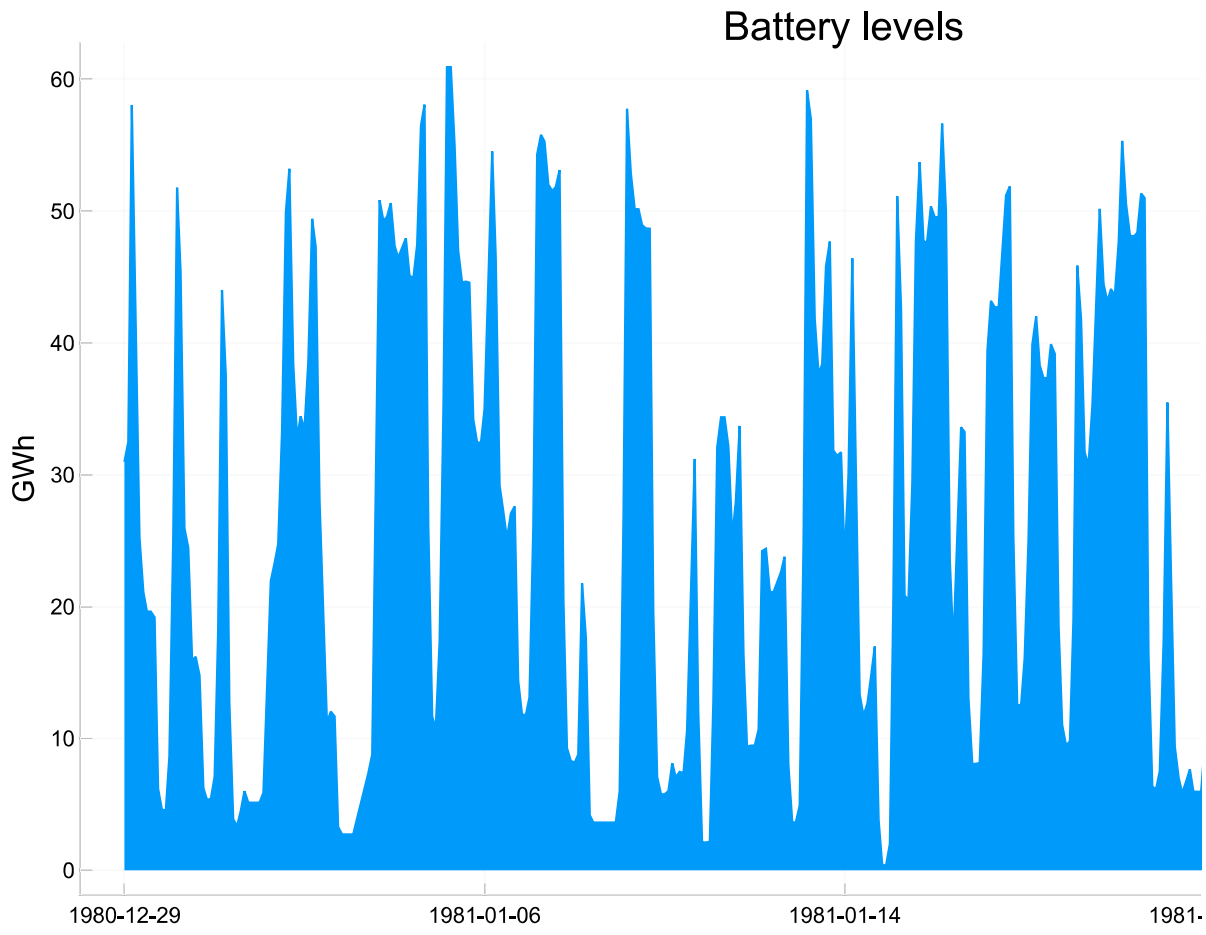
In [22]:

```
# Plot prices
idxwohub = findall(x -> !occursin("HUB", x), powerbalancenames) # remove hubs, not
display(plot(x1, prices[:,idxwohub], labels=reshape(powerbalancenames[idxwohub],1,

# # Plot supplies and demands
# supplychart = plot(x1, supplyvalues,labels=reshape(supplynames,1,length(supplynar
# demandchart = plot(x1, demandvalues,labels=reshape(demandnames,1,length(demandnar
supplychart = areaplot(x1, sum(supplyvalues,dims=2),title="Supply", ylabel = "GWh/
demandchart = areaplot(x1, sum(demandvalues,dims=2),title="Demand", ylabel = "GWh/
display(plot([supplychart,demandchart]...,layout=(1,2),size=(800,500)))

# Plot storages
# display(areaplot(x2, hydrolevels1,labels=reshape(hydronames,1,length(hydronames),
display(areaplot(x2, dropdims(sum(hydrolevels,dims=2),dims=2),labels="Total reserv

display(areaplot(x1, dropdims(sum(batterylevels,dims=2),dims=2),labels="Total batt

# Plot list of yearly mean production and demand for each supply/demand
meandemand = dropdims(mean(demandvalues,dims=1),dims=1)
meanproduction = dropdims(mean(supplyvalues,dims=1),dims=1)
supplydf = sort(DataFrame(Supplyname = supplynames, Yearly_supply_TWh = meanproduc
demanddf = sort(DataFrame(Demandname = demandnames, Yearly_demand_TWh = meandemand
supplydf[!,:ID] = collect(1:length(supplynames))
demanddf[!,:ID] = collect(1:length(demandnames))
joineddf = select!(outerjoin(supplydf,demanddf;on=:ID),Not(:ID))
# show(joineddf,allcols=true, allrows=true, nosubheader = true)

# Check that total supply equals total demand
show(combine(joineddf, [:Yearly_supply_TWh, :Yearly_demand_TWh] .=> sum∘skipmissing

# # Plot list of yearly income and cost for each supply/demand (only works if exoge
# supplyrev = copy(supplyvalues)
# for (i,supplybalancename) in enumerate(supplybalancenames)
#     idx = findfirst(isequal(supplybalancename), powerbalancenames)
#     supplyrev[:,i] .= supplyrev[:,i] .* prices[:,idx]
# end
# demandrev = copy(demandvalues)
# for (i,demandbalancename) in enumerate(demandbalancenames)
#     idx = findfirst(isequal(demandbalancename), powerbalancenames)
#     demandrev[:,i] .= demandrev[:,i] .* prices[:,idx]
# end
# meandemandrev = dropdims(mean(demandrev,dims=1),dims=1)
# meanproductionrev = dropdims(mean(supplyrev,dims=1),dims=1)
# supplyrevdf = sort(DataFrame(Supplyname = supplynames, Yearly_rev_mill€ = meanpro
# demandrevdf = sort(DataFrame(Demandname = demandnames, Yearly_cost_mill€ = meande
# supplyrevdf[!,:ID] = collect(1:length(supplynames))
# demandrevdf[!,:ID] = collect(1:length(demandnames))
# joinedrevdf = select!(outerjoin(supplyrevdf,demandrevdf;on=:ID),Not(:ID))
# # show(joinedrevdf,allcols=true, allrows=true, nosubheader = true)

# # Sum revenues and cost
# show(combine(joinedrevdf, [:Yearly_rev_mill€, :Yearly_cost_mill€] .=> sum∘skipmis
```


Prices
€/MWh
0.8
0.6
0.4
0.2
0.0
1980-12-29
1981-01-06
1981-01-14
1981


Supply
D
GWh/h
500
400
300
200
100
0
1980-12-29
1981-01-06
1981-01-14
1981-01-22
1981-01-30
GWh/h
500
400
300
200
100
0
1980-12-29
1981-01-06


Reservoir levels
TWh
80
60
40
20
0
1980-12-29
1981-01-06
1981-01-14
1981


Battery levels
GWh
60
50
40
30
20
10
0
1980-12-29
1981-01-06
1981-01-14
1981-

```
1×2 DataFrame
 Row │ Yearly_supply_TWh_sum_skipmissing  Yearly_demand_TWh_sum_skipmissing
     │
```

- Yearly_supply_TWh and Yearly_demand_TWh is the mean production timed with 8736.
- Transmission into the area is added to supply, while transmission out is added to demand.
- The reason why the supply does not match the demand is that the filtering does not split up watercourses where hydropower plants are in different areas. The supply side (of NO2) therefore contains 14 hydropower plants in NO1 and NO5.

In [ ]: