Import packages and code

We build the simulation model with TuLiPa and use some functionality from JulES

In [18]:
```
using DataFrames, Plots, Statistics, JSON, Distributed, Distributions
plotlyjs()
include(joinpath(dirname(pwd()),"src/TuLiPa.jl"))
include(joinpath(dirname(dirname(pwd())),"JulES/src/JulES.jl"));
```

Demo 9 - Hydro simulation w head dependence and scenario modelling

Demo 9 solves a two stage stochastic LP in the simulation steps to optimize the Fykan (Svartisen) watercourse against an exogen area. The two stage stochastic LP is decomposed with Benders. New from the previous demos is that:

- We use the Hydrocen 2030 low emission price series (TODO: Also use for previous demos)
- We consider head dependence in the master problem of the Benders decomposition. Firstly we set state dependent energy equivalents for plants and pumps (variable based on head), and secondly we calculate head loss costs for some reservoirs to increase the watervalue of reservoirs that would give a higher production with a higher reservoir filling.
  - Necessary metadata like reservoir curves and nominal head of hydropower plants has been added to the dataset to be able to do these calculations.
  - Head dependency is more visible for some watercourses. Set boolheadlosscosts = false for Svartisen (watercourse = "FYKAN_H") to see the difference it makes.
- 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 demo we use InflowClusteringMethod.

Starting point and scenario settings

- Start model year 2021 (dataset only consist of hydro system from 2021 and prices from 2030) and weather scenario 1981
- 30 possible scenarios that can be used to consider uncertainty (we use scenario modelling later)
- Series simulation 30 years with moving horizons and two day time steps. In each time step a two stage stochastic programming problem is solved with Benders decomposition. First stage (master) problem is 2 days long and scenario problems looks 4 years into the future. Here we have a long horizon since we are looking at a watercourse with a multi-year reservoir.
- Scenarios are phased in after two days
  - First two days (master problem) have perfect information from simulation/main scenario
  - Next 5 weeks combines simulation/main scenario and scenario X. Smooth transition in 5 steps.
  - After 5 weeks the simulation/main scenario is fully phased out.

In [2]:
```
datayear = 2021
scenarioyearstart = 1981
scenarioyearstop = 2010
totalscen = 30 # scenarios to consider uncertainty for
numscen = 7 # scenario reduction to this amount
totalduration = Millisecond(Day(364*4)) # length of master + subproblem
mpdp = Millisecond(Hour(24)) # period duration for different commodities can differ, also between the master and sub problems
mpdh = Millisecond(Hour(24)) # master/sub - period duration - commodity (power, hydro)
spdp = Millisecond(Hour(168))
spdh = Millisecond(Hour(168))

# Time structure used in master problem (simple since no phasein)
tnormal = FixedDataTwoTime(getisoyearstart(datayear),getisoyearstart(scenarioyearstart))

# Phasein parameters
phaseinoffsetdays = 2 # also master problem length
phaseinoffset = Millisecond(Day(phaseinoffsetdays)) # phase in straight away from second stage scenarios
phaseindelta = Millisecond(Week(5)) # Phase in the second stage scenario over 5 weeks
phaseinsteps = 7 # Phase in second stage scenario in 5 steps

timeresolutioninfo = (totalduration, phaseinoffset, mpdp, mpdh, spdp, spdh);

# Make scenario times for all uncertainty scenarios
totalscentimes = []
for scen in 1:totalscen
    scentnormal = FixedDataTwoTime(getisoyearstart(datayear),getisoyearstart(scenarioyearstart + scen - 1))
    scentphasein = PhaseinTwoTime(getisoyearstart(datayear),getisoyearstart(scenarioyearstart), getisoyearstart(scenarioyearstart +

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

Make and load dataset

You can choose from the 125 Norwegian watercourses in TuLiPa/demos/dataset_vassdrag.

And the 11 Hydrocen prices from different Norwegian EMPS areas:

- Ostland, Sorost, Hallingdal, Telemark, Sorland, Vestsyd, Vestmidt, Norgemidt, Helgeland, Troms, Finnmark
- A map to locate the EMPS areas (NVE has 15 EMPS areas so a bit different)
- Possible mapping of EMPS areas to elspot areas: NO1 -> Ostland, NO2 -> Sorland, NO3 -> Norgemidt, NO4 -> Troms, NO5 -> Vestsyd

In [3]:
```
# watercourse = "AURLAND_H"
# elspotnames = ["NO5"] # some watercourses are in several elspot areas
```

```
# priceseriesname = "Vestsyd"

# Svartisen is a good example of a watercourse with high head effects. Try setting boolheadlosscosts = false to see the difference it
watercourse = "FYKAN_H"
elspotnames = ["NO4"] # some watercourses are in several elspot areas
priceseriesname = "Troms"

# Read hydropower dataelements from json-files
sti_dynmodelldata = "dataset_vassdrag"
tidsserie = JSON.parsefile(joinpath(sti_dynmodelldata,"tidsserier_detd.json"))
detdseries = getelements(tidsserie, sti_dynmodelldata);
dst = JSON.parsefile(joinpath(sti_dynmodelldata, "dataset_detd_" * watercourse * ".json"))
detdstructure = getelements(dst);
elements = vcat(detdseries,detdstructure)

# Add an exogenous price area that the plants and pumps can interact with. All units are in NO5.
for elspotname in elspotnames
    addexogenbalance!(elements, "PowerBalance_" * elspotname, "Power", "AreaPrice")
end

# Add dataelements for price in exogen area
file = "Prices_HydroCen_LowEmission_2030_1958-2015.csv"
df = CSV.read(file, DataFrame; header=3, decimal=',', types=Float64)
df[:,"aar"] = cld.(1:first(size(df)), 2912) .+ 1957
df[:,"tsnitt"] = rem.(0:(first(size(df))-1), 2912) .+ 1
df.datetime .= getisoyearstart.(Int.(df.aar)) + Hour.((df.tsnitt.-1)*3) # TODO: Include week 53. Now ignored and flat prices.
push!(elements, DataElement(TIMEINDEX_CONCEPT,"VectorTimeIndex","AreaPriceProfileIndex",
        Dict("Vector" => df.datetime)))
push!(elements, DataElement(TIMEVALUES_CONCEPT,"VectorTimeValues","AreaPriceProfileValues",
        Dict("Vector" => df[:,priceseriesname])))
push!(elements, getelement(TIMEVECTOR_CONCEPT,"RotatingTimeVector","AreaProfile",
        (TIMEINDEX_CONCEPT,"AreaPriceProfileIndex"),(TIMEVALUES_CONCEPT,"AreaPriceProfileValues")))
addparam!(elements, "MeanSeriesIgnorePhaseinParam", "AreaPrice", 1.0, "AreaProfile")

# Select which scenarios to include from the time-series
addscenariotimeperiod!(elements, "ScenarioTimePeriod", getisoyearstart(scenarioyearstart), getisoyearstart(scenarioyearstop));
```

Make master and subobjects

The starting point is the dataelements that represent the watercourse and exogen price area. Then we add the time resolution info and number of scenarios to build different model objects for the master and subproblems.

In [4]:
```
# Make master or subobjects given time resolution info
function singlemakestochasticobjects(elements, duration, offset, pdp, pdh)

    # Add horizons to elements
    power_horizon = SequentialHorizon(ceil(Int64, duration/pdp), pdp; offset)
    hydro_horizon = SequentialHorizon(ceil(Int64, duration/pdh), pdh; offset)

    push!(elements, getelement(COMMODITY_CONCEPT, "BaseCommodity", "Power",
            (HORIZON_CONCEPT, power_horizon)))
    push!(elements, getelement(COMMODITY_CONCEPT, "BaseCommodity", "Battery",
            (HORIZON_CONCEPT, power_horizon)))
    push!(elements, getelement(COMMODITY_CONCEPT, "BaseCommodity", "Hydro",
            (HORIZON_CONCEPT, hydro_horizon)))

    # Make modelobjets from elements
    modelobjects = getmodelobjects(elements)

    return collect(values(modelobjects))
end

# Make master and subobjects for different scenarios
function singlemakemastersubobjects!(inputs)
    (elements, numscen, timeresolutioninfo) = inputs
    (totalduration, phaseinoffset, mpdp, mpdh, spdp, spdh) = timeresolutioninfo

    masterobjects = singlemakestochasticobjects(copy(elements), phaseinoffset, nothing, mpdp, mpdh)

    subscenarioobjects = []
    for scenario in 1:numscen # as long as this is not run it parallel we actually only need one subproblem that can be updated for d
        offset = TimeDeltaOffset(MsTimeDelta(phaseinoffset))
        push!(subscenarioobjects, singlemakestochasticobjects(copy(elements), totalduration - phaseinoffset, offset, spdp, spdh))
    end

    return (masterobjects, subscenarioobjects)
end

inputs = (elements, numscen, timeresolutioninfo)
(masterobjects, subobjects) = singlemakemastersubobjects!(inputs);
```

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 demo we use InflowClusteringMethod which cluster together scenarios with similar energy inflows (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:

- 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)

In [5]:
```
parts = 4 # divide scendelta into this many parts, calculate sum inflow for each part of the inflow series, then use clustering algor
scenmodmethod = InflowClusteringMethod(numscen, parts) # see JulES/scenariomodelling.jl

# maxquantile = 0.99
# a = 0.5
# b = 4
# c = 10
# a = 1
# b = 0
# c = 5
# a = 0
# b = 0
# c = 1
# x = collect(-numscen+1:2:numscen-1)
# y = a .* x .^ 2 .+ x .* b .+ c
# display(y/sum(y)) # show chosen weights
# scenmodmethod = SumInflowQuantileMethod(numscen, maxquantile, a, b, c)
# scenmodmethod = SumInflowQuantileMethod(numscen, maxquantile, a, b, c, usedensity=true)

# scenmodmethod = NoScenarioModellingMethod(numscen, totalscentimes)

scendelta = MsTimeDelta(Day(364)) # scenario modelling based on the next year, even though the scenario problems can be longer
scenariomodelling!(scenmodmethod, subobjects[1], numscen, totalscentimes, scendelta) # see JulES/scenariomodelling.jl
# - scenmodmethod = scenario modelling method
# - subobjects[1] = objects to do scenario modelling on, will here use energy inflow from these objects
# - numscen = number of clusters and scenarios to end up with
# - totalscentimes = all scenarios that are reducted to fewer scenarios
# - scendelta = # scenario modelling based on this part of the time period in the horizon
```

Initialize master and subproblems and run first iteration

In [6]:
```
# Initialize stochastic subsystem problems and solve for first time step
function singlestochastic_init(masterobjects, subobjects, startstorage, lb, maxcuts, reltol, t, scenmodmethod::ScenarioModellingMeth

    # Initialize cuts
    cutobjects = getcutobjects(masterobjects)
    cuts = initialize_cuts!(masterobjects, cutobjects, maxcuts, lb, length(subobjects))
    cuts.probabilities = scenmodmethod.weights
    states = getstatevariables(cutobjects) # state variables in master and subs for boundary reservoirs

    # master = JuMP_Prob(masterobjects, Model(HiGHS.Optimizer))
    # subs = [JuMP_Prob(subobject, Model(HiGHS.Optimizer)) for subobject in subobjects] # initialize subproblems
    master = HiGHS_Prob(masterobjects)
    subs = [HiGHS_Prob(subobject) for subobject in subobjects] # initialize subproblems

    # State dependent hydropower prod and pumping
    statedependentprod_init!(master, startstorage, t)
    statedependentpump_init!(master, startstorage, t)

    # Update master
    update!(master, t)
    setstartstoragepercentage!(master, getstorages(getobjects(master)), t, startstorage)

    # Get price and use as end value (assumes all reservoirs are in the same price area)
    # TODO: Replace with function that searches the watercourse for the closest plant/pump for each reservoir
    exogenprice = findfirstprice(master.objects)

    # Init cutparameters
    cutparameters = Vector{Tuple{Float64, Dict{StateVariableInfo, Float64}}}(undef, length(subs)) # preallocate for cutparameters fr

    # Update subs
    for (i,sub) in enumerate(subs)
        scaleinflow!(scenmodmethod, i, sub.objects)

        (scentnormal,scentphasein, scen) = scenmodmethod.scentimes[i]
        update!(sub, scentphasein) # update parameters given problem start time of scenario

        storages = getstorages(getobjects(sub))
        # Alternative end reservoir conditions:
        # - Stop equals start reservoir, and long enough horizon (can crash if many start reservoirs full or empty)
        # - End value = 0, and long enough horizon
        # - List of endvalues from other calculation, maybe SDP
        # - End value equals average price in this price area for different scenarios (could also adjust based on reservoir flexibili

        # setendstoragepercentage!(sub, storages, scentphasein, startstorage) # set end reservoir

        # Set monthly price at end of horizon as end value
        scenprice = getparamvalue(exogenprice, scentnormal + getduration(gethorizon(storages[1])), MsTimeDelta(Week(4)))
        subendvaluesid = Id(BOUNDARYCONDITION_CONCEPT,"EndValue")
        subendvaluesobj = EndValues(subendvaluesid, storages)
        push!(sub.objects, subendvaluesobj)
        subendvalues = scenprice .* [getbalance(obj).metadata[GLOBALENEQKEY] for obj in storages] # end value also depends on the en
        updateendvalues!(sub, subendvaluesobj, subendvalues)
    end

    ub = 0
    cutreuse = false # reuse cuts in first iteration of Benders?
    iterate_convergence!(master, subs, cuts, cutparameters, states, cutreuse, lb, ub, reltol) # Benders iteration
```

```
    return (master, subs, states, cuts, cutparameters, exogenprice)
end

# Find first exogen price in a vector of model objects
function findfirstprice(objects)
    for obj in objects
        if obj isa ExogenBalance
            return getprice(obj)
        end
    end
end;
```

In [7]:
```
# Cut parameters
maxcuts = 13 # preallocate fixed number of cuts, no cut selection
lb = -1e10 # lower bound of the future value in the first iteration
# lb = -1e5 # lower bound of the future value in the first iteration
reltol = 0.0001 # relative tolerance

# Percentage start storage
startstorage = 65

(master, subs, states, cuts, cutparameters, exogenprice) = singlestochastic_init(masterobjects, subobjects, startstorage, lb, maxcut
```

Collect watervalues from scenarios and operative water values (also headloss adjusted operative water value further down)

In [8]:
```
masterobjects = Dict(zip([getid(obj) for obj in master.objects],master.objects))
watervalues = zeros(1, numscen + 2, length(cuts.objects))
for i in 1:numscen
    (constant, slopes) = cutparameters[i]
    for (j, (state, value)) in enumerate(cuts.slopes[1])
        watervalues[1, i, j] = slopes[state] / getbalance(masterobjects[first(getvarout(state))]).metadata[GLOBALENEQKEY]
    end
end
for (j,obj) in enumerate(cuts.objects) # operative water values
    balance = getbalance(obj)
    watervalues[1, numscen + 1, j] = getcondual(master, getid(balance), getnumperiods(gethorizon(balance))) / balance.metadata[GLOBA
end
```

Run master problem again with headloss cost

We use the 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 it will be impacted a lot by the headloss cost.

See the code at TuLiPa/reasoning_nvehydro.jl

In [9]:
```
boolheadlosscosts = true
if boolheadlosscosts
    updateheadlosscosts!(ReservoirCurveSlopeMethod(), master, [master], tnormal)
    solve!(master)
    resetheadlosscosts!(master)
end
```

Collect end state variables for use as start variables in next time step

In [10]:
```
startstates_ = getstatevariables(master.objects)
getoutgoingstates!(master, startstates_)
startstates = Dict{String, Float64}()
for var in keys(startstates_)
    startstates[getinstancename(first(getvarout(var)))] = startstates_[var]
end
```

Collect results from masterproblem

In [11]:
```
resultobjects = master.objects # collect results for all areas
prices, rhstermvalues, production, consumption, hydrolevels, batterylevels, powerbalances, rhsterms, rhstermbalances, plants, plantb

for (j,obj) in enumerate(cuts.objects) # operative water value after headloss adjustment
    balance = getbalance(obj)
    watervalues[1, numscen + 2, j] = getcondual(master, getid(balance), getnumperiods(gethorizon(balance))) / balance.metadata[GLOBA
end
```

Simulate next time steps

- Simulate next time steps, store results and plot results

In [12]:
```
# Similar but without updating prices
function singlestochastic!(master, subs, states, cuts, cutparameters, startstates, reltol, t, exogenprice, scenmodmethod)
    # Update probabilities in cuts
    cuts.probabilities = scenmodmethod.weights

    # Update master
```

```
    masterstorages = getstorages(master.objects)
    setstartstates!(master, masterstorages, startstates)

    # Statedependent hydropower production and pumping
    statedependentprod!(master, startstates)
    statedependentpump!(master, startstates)

    update!(master, t)

    # Update subs
    for (i,sub) in enumerate(subs)

        scaleinflow!(scenmodmethod, i, sub.objects)

        (scentnormal,scentphasein, scen) = scenmodmethod.scentimes[i]
        update!(sub, scentphasein) # update parameters given problem start time of scenario

        storages = getstorages(getobjects(sub))
        # setendstates!(sub, substorages, startstates) # set end reservoir

        # Set monthly price at end of horizon as end value
        scenprice = getparamvalue(exogenprice, scentnormal + getduration(gethorizon(storages[1])), MsTimeDelta(Week(4)))
        subendvaluesobj = sub.objects[findfirst(x -> getid(x) == Id(BOUNDARYCONDITION_CONCEPT,"EndValue"), sub.objects)]
        subendvalues = scenprice .* [getbalance(obj).metadata[GLOBALENEQKEY] for obj in storages] # end value also depends on the en
        updateendvalues!(sub, subendvaluesobj, subendvalues)
    end

    lb = cuts.lower_bound
    ub = 0
    cutreuse = true
    iterate_convergence!(master, subs, cuts, cutparameters, states, cutreuse, lb, ub, reltol)
end;
```

In [13]:
```
# How many steps to simulate?
# steps = 16;
steps = (getisoyearstart(scenarioyearstop) - getisoyearstart(scenarioyearstart)).value/phaseinoffset.value;
```

In [14]:
```
# Increment steps and scenariotimes
step = 2
tnormal += Day(phaseinoffsetdays)
for scen in 1:totalscen
    (scentnormal, scentphasein, scen) = totalscentimes[scen]
    scentnormal += Day(phaseinoffsetdays)
    scentphasein = PhaseinTwoTime(getdatatime(tnormal), getscenariotime(tnormal), getscenariotime(scentnormal), phaseinoffset, phase

    totalscentimes[scen] = (scentnormal, scentphasein, scen)
end
# display(tnormal)

totaltime = @elapsed while step <= steps

    # Scenariogeneration for subproblems
    scenariomodelling!(scenmodmethod, subs[1].objects, numscen, totalscentimes, scendelta)

    # Stochastic sub systems - calculate storage value
    singlestochastic!(master, subs, states, cuts, cutparameters, startstates, reltol, tnormal, exogenprice, scenmodmethod)

    # Collect watervalues from scenarios and operative (also headloss adjusted water value further down)
    newwatervalues = zeros(1, numscen + 2, length(cuts.objects)) # watervalues in last cut
    for i in 1:numscen
        (constant, slopes) = cutparameters[i]
        for (j, (state, value)) in enumerate(cuts.slopes[1])
            newwatervalues[1, i, j] = slopes[state] / getbalance(masterobjects[first(getvarout(state))]).metadata[GLOBALENEQKEY]
        end
    end
    for (j,obj) in enumerate(cuts.objects) # operative water value
        balance = getbalance(obj)
        newwatervalues[1, numscen + 1, j] = getcondual(master, getid(balance), getnumperiods(gethorizon(balance))) / balance.metadat
    end

    # Rerun master problem with headloss costs
    if boolheadlosscosts
        updateheadlosscosts!(ReservoirCurveSlopeMethod(), master, [master], tnormal)
        solve!(master)
        resetheadlosscosts!(master)
    end

    # Update startstates
    startstates_ = getstatevariables(master.objects)
    getoutgoingstates!(master, startstates_)
    for var in keys(startstates_)
        # value = round(startstates_[var], digits=10) # avoid approx 0 negative values, ignored by solvers so no problem?
        startstates[getinstancename(first(getvarout(var)))] = startstates_[var]
    end

    # Collect results
    prices, rhstermvalues, production, consumption, hydrolevels, batterylevels = update_results(master, prices, rhstermvalues, produ
    for (j,obj) in enumerate(cuts.objects) # operative water value after headloss adjustment
        balance = getbalance(obj)
        newwatervalues[1, numscen + 2, j] = getcondual(master, getid(balance), getnumperiods(gethorizon(balance))) / balance.metadat
    end
    watervalues = vcat(watervalues, newwatervalues)

    # Increment steps and scenariotimes
    step += 1
```

```
    tnormal += Day(phaseinoffsetdays)
    for scen in 1:totalscen
        (scentnormal, scentphasein, scen) = totalscentimes[scen]
        scentnormal += Day(phaseinoffsetdays)
        scentphasein = PhaseinTwoTime(getdatatime(tnormal), getscenariotime(tnormal), getscenariotime(scentnormal), phaseinoffset, pl

        totalscentimes[scen] = (scentnormal, scentphasein, scen)
    end
    # display(tnormal)
end

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

```
"The simulation took: 6.438394253333333 minutes"
"Time usage per timestep: 0.0729012370635969 seconds"
```

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 [15]:

```
# Only keep rhsterms that have at least one value (TODO: Do the same for sypply and demands)
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[rhstermsupplyidx]]
rhstermsupplybalancenames = [split(getinstancename(r), "PowerBalance_")[2] for r in rhstermbalances[rhstermsupplyidx]]
rhstermdemandnames = [getinstancename(rhsterm) for rhsterm in rhsterms[rhstermdemandidx]]
rhstermdemandbalancenames = [split(getinstancename(r), "PowerBalance_")[2] for r in rhstermbalances[rhstermdemandidx]]

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

demandnames = [[getinstancename(demand) for demand in demands];rhstermdemandnames]
demandbalancenames = [[split(getinstancename(p), "PowerBalance_")[2] for p in demandbalances];rhstermdemandbalancenames]
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_")[2] for powerbalance in powerbalances]

# Convert reservoir filling to TWh
hydrolevels1 = copy(hydrolevels)
for (i,hydroname) in enumerate(hydronames)
    hydrolevels1[:,i] = hydrolevels1[:,i]*getbalance(masterobjects[hydrostorages[i]]).metadata[GLOBALENEQKEY]
end

# Time
x1 = [getisoyearstart(scenarioyearstart) + mpdp*(t-1) for t in 1:first(size(supplyvalues))] # power/load resolution
x2 = [getisoyearstart(scenarioyearstart) + mpdh*(t-1) for t in 1:first(size(hydrolevels))]; # reservoir resolution
```

In [16]:

```
# areaprices = rename!(DataFrame(prices, :auto),powerbalancenames)
# areaprices[!,:time] = x1
# CSV.write(folder * "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(folder * "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(folder * "supply.csv", supply)

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

# battery = rename!(DataFrame(batterylevels, :auto),batterynames)
# battery[!,:time] = x1
# CSV.write(folder * "batteries.csv", battery);
```

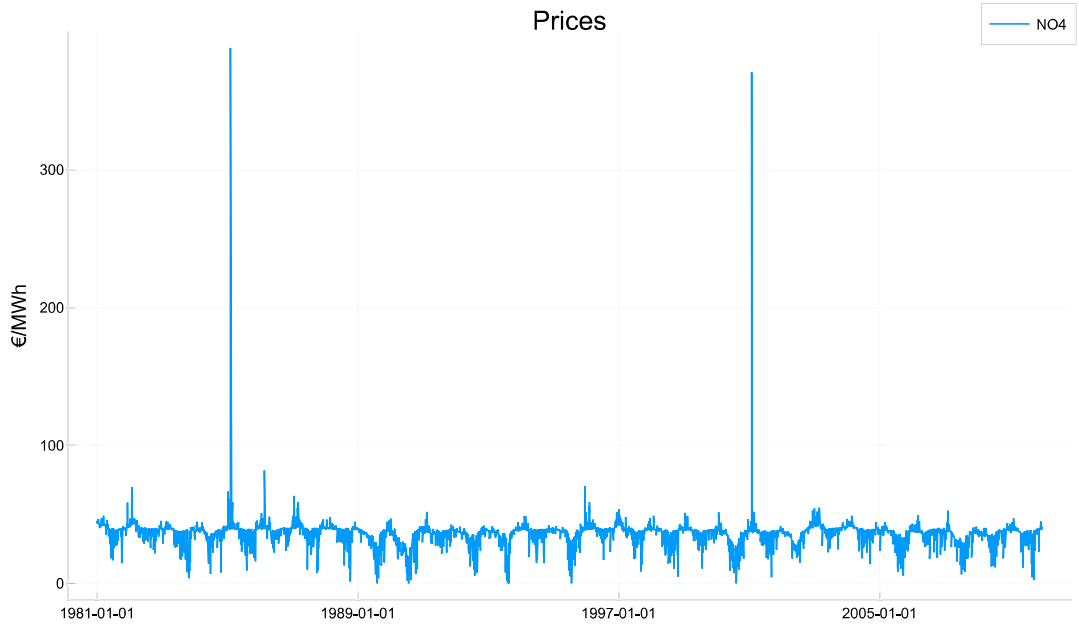
Plot results

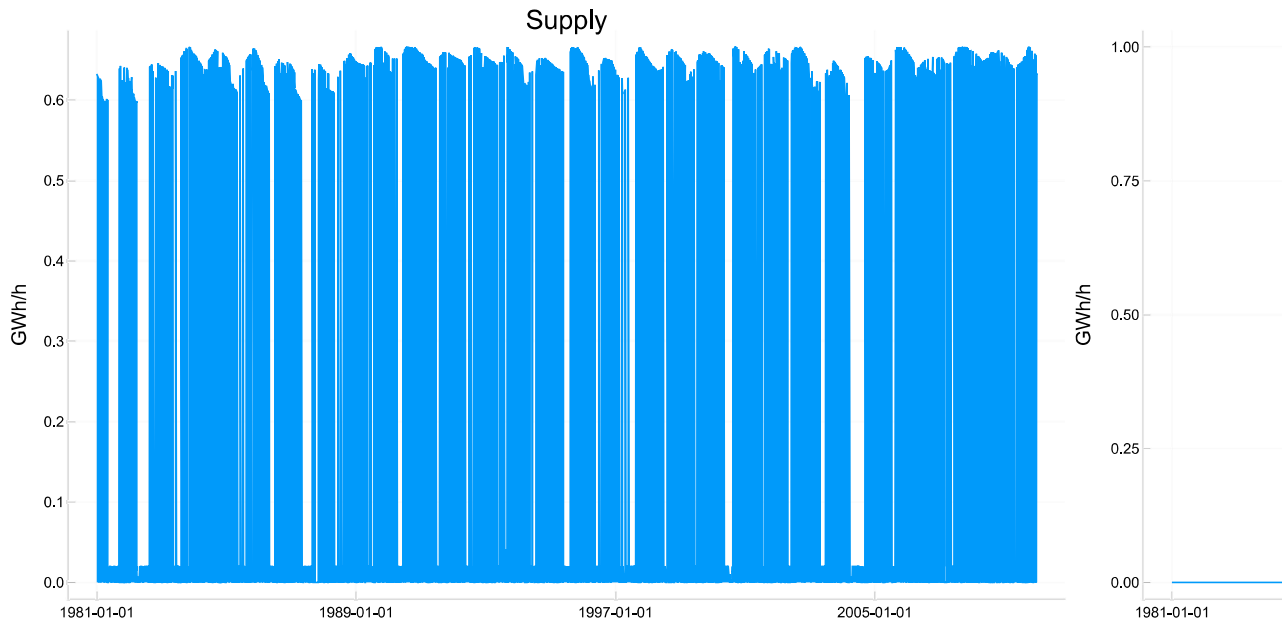
Notice:

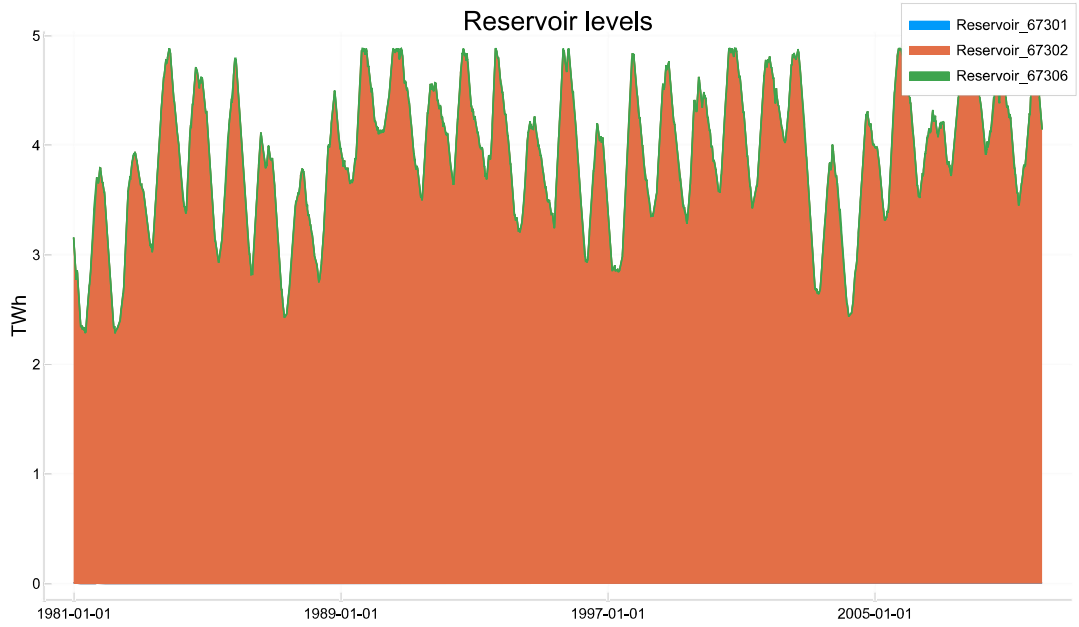
- That the max production now depends on the reservoir filling (head dependent production)
- The reservoir filling is a lot higher due to the headloss cost (try setting boolheadlosscosts = false to see the difference). This gives a higher revenue when the plant produces at a higher head.
- For reservoir 67302 the operative water value after headloss adjustment is a lot higher than before adjustment. This incentivises the model to keep the reservoir filling high.

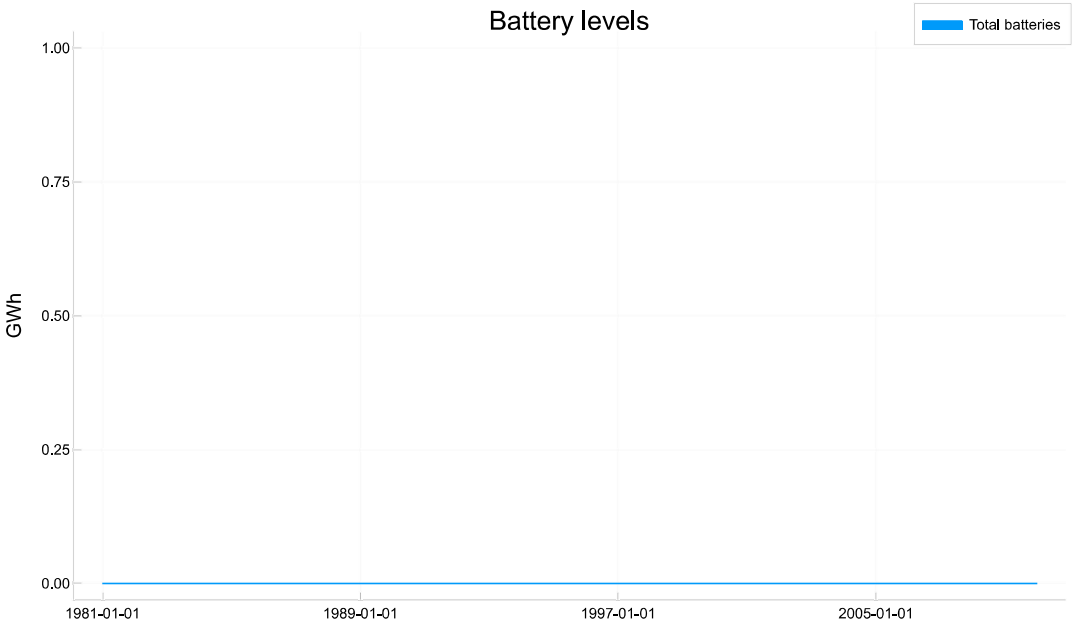
In [17]:

```
# Plot prices
display(plot(x1, prices, labels=reshape(powerbalancenames,1,length(powerbalancenames)), size=(800,500), title="Prices", ylabel="€/MWh

# # Plot supplies and demands
# supplychart = plot(x1, supplyvalues,labels=reshape(supplynames,1,length(supplynames)),title="Supply", ylabel = "GWh/h")
# demandchart = plot(x1, demandvalues,labels=reshape(demandnames,1,length(demandnames)),title="Demand", ylabel = "GWh/h")
supplychart = areaplot(x1, sum(supplyvalues,dims=2),title="Supply", ylabel = "GWh/h")
demandchart = areaplot(x1, sum(demandvalues,dims=2),title="Demand", ylabel = "GWh/h")
display(plot([supplychart,demandchart]...,layout=(1,2),size=(1600,500)))

# Plot storages
display(areaplot(x2, hydrolevels1,labels=reshape(hydronames,1,length(hydronames)),size=(800,500),title="Reservoir levels", ylabel = "
# display(areaplot(x2, dropdims(sum(hydrolevels,dims=2),dims=2),labels="Total reservoirs",size=(800,500),title="Reservoir levels", yl

display(areaplot(x1, dropdims(sum(batterylevels,dims=2),dims=2),labels="Total batteries",size=(800,500),title="Battery levels", ylabe

# 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 = meanproduction*8.76),[:Yearly_supply_TWh], rev = true)
demanddf = sort(DataFrame(Demandname = demandnames, Yearly_demand_TWh = meandemand*8.76),[:Yearly_demand_TWh], rev = true)
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), nosubheader = true)

# Plot list of yearly income and cost for each supply/demand
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€ = meanproductionrev*8.76),[:Yearly_rev_mill€], rev = true)
demandrevdf = sort(DataFrame(Demandname = demandnames, Yearly_cost_mill€ = meandemandrev*8.76),[:Yearly_cost_mill€], rev = true)
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∘skipmissing), nosubheader = true)

# Plot watervalues for each reservoir and scenarios
for (i,obj) in enumerate(cuts.objects)
    name = getinstancename(getid(obj))

   p = plot(watervalues[:,1:numscen,i] * -1, size=(800,500), title="Watervalues scenario and operative for " * name, ylabel="€/MWh")
   plot!(p, watervalues[:,numscen+1,i] * -1, labels="Operative before")
   plot!(p, watervalues[:,numscen+2,i] * -1, labels="Operative after")
   display(p)
end
```


Prices
NO4
€/MWh
300
200
100
0
1981-01-01
1989-01-01
1997-01-01
2005-01-01


Supply
GWh/h
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1981-01-01
1989-01-01
1997-01-01
2005-01-01
GWh/h
1.00
0.75
0.50
0.25
0.00
1981-01-01


Reservoir levels
Reservoir_67301
Reservoir_67302
Reservoir_67306
TWh
5
4
3
2
1
0
1981-01-01
1989-01-01
1997-01-01
2005-01-01



2×4 DataFrame

Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
1	Release_67302	2.30758	missing	missing
2	Release_67301	0.0718053	missing	missing

1×2 DataFrame

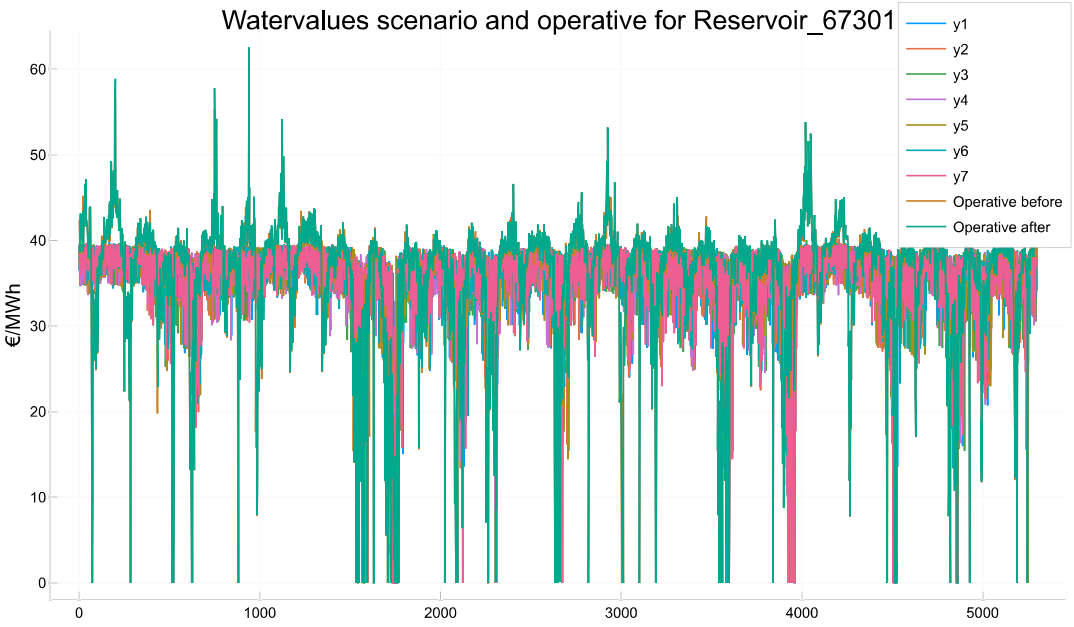
Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing
1	2.37939	0.0

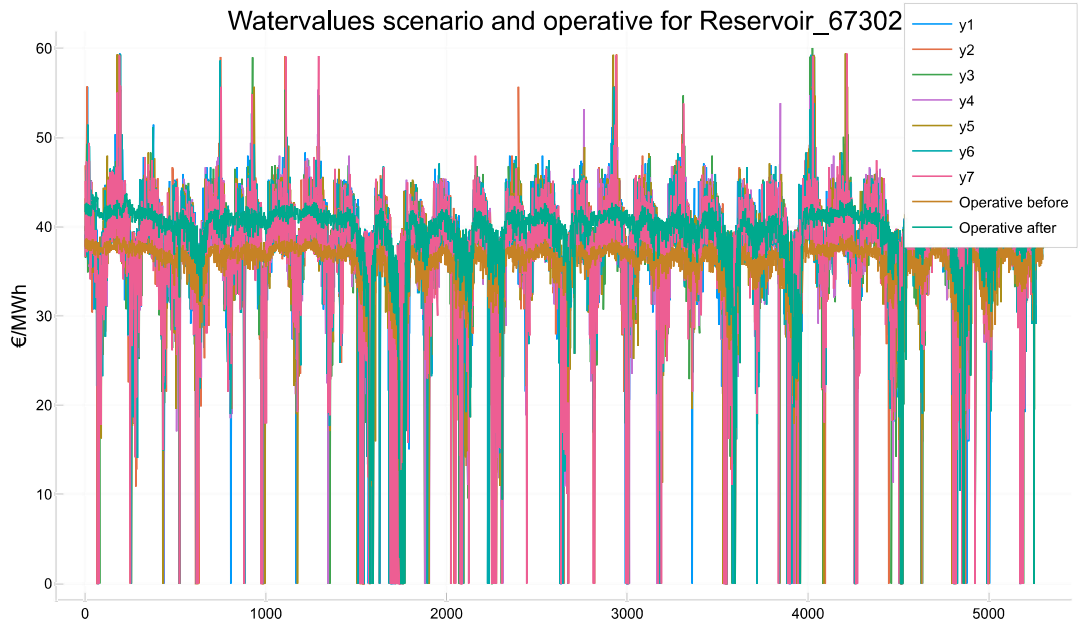
2×4 DataFrame

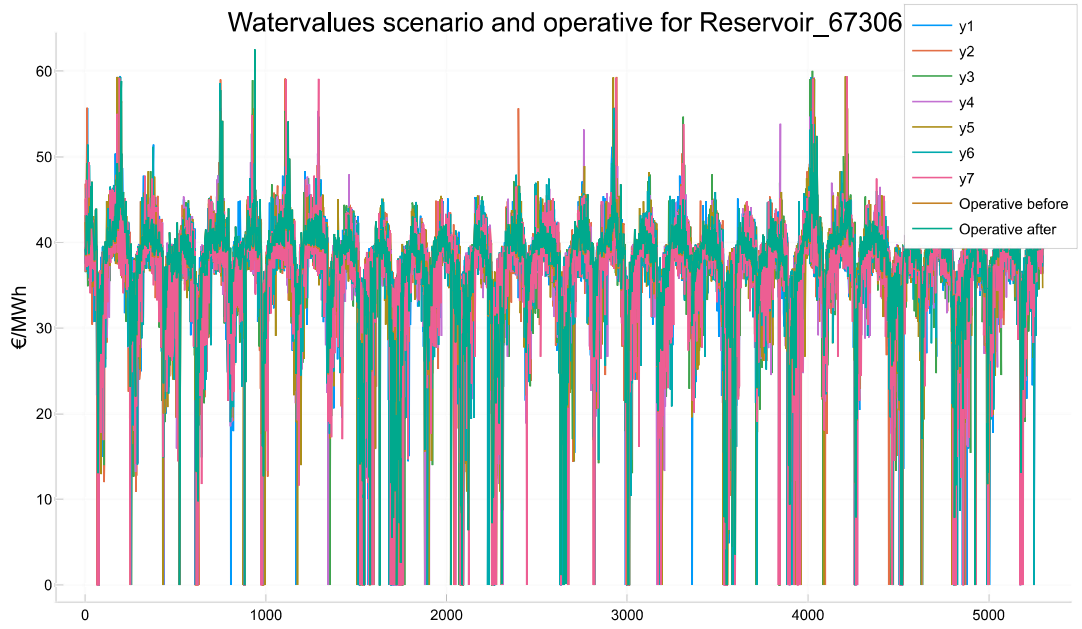
Row	Supplyname	Yearly_rev_mill€	Demandname	Yearly_cost_mill€
1	Release_67302	89.8475	missing	missing
2	Release_67301	2.6509	missing	missing

1×2 DataFrame

Row	Yearly_rev_mill€_sum_skipmissing	Yearly_cost_mill€_sum_skipmissing
1	92.4984	0.0




Watervalues scenario and operative for Reservoir_67302
€/MWh
0
10
20
30
40
50
60
0
1000
2000
3000
4000
5000
y1
y2
y3
y4
y5
y6
y7
Operative before
Operative after


Watervalues scenario and operative for Reservoir_67306
€/MWh
0
10
20
30
40
50
60
0
1000
2000
3000
4000
5000
y1
y2
y3
y4
y5
y6
y7
Operative before
Operative after