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

In [2]:
```
include(joinpath(dirname(pwd()),"src/TuLiPa.jl"));
```

In [3]:
```
include(joinpath(dirname(dirname(dirname(pwd()))),"datasett/JulES/src/JulES.jl"));
```

Demo 8 - Solar & Battery subsystem simulation w Benders

Demo 8 simulates a solar plant and a battery against an exogen area. The solar production can either be used to charge the battery or can be sold to the market. The battery can also exchange power with the power market.

The simulation algorithm is a fan simulator that solves a two-stage stochastic optimization problem with Benders decomposition for each time step. The time steps and master problems are 2 days long, and the scenario problems are 5 days long and consider uncertainty from 15 weather scenarios.

The demo also shows how to build the dataset. We use the solar profile from https://www.nve.no/energi/analyser-og-statistikk/vaerdatasett-for-kraftsystemmodellene/. The price for the exogen area is generated in 1.6 in Demo 2 (TODO: replace price with HydroCen price series).

Starting point and scenario settings

- Start point model year 2025 and weather scenario 1981
- Series simulation over 15 years with moving horizons and two day time steps, and 5 days long scenario problems.
- Hourly time resolution for power market and battery
- 15 weather years considered for uncertainty
- Scenarios are phased in after two days
  - First two days have perfect information from starting point scenario
  - Next 5 days combines starting point scenario and scenario X. Smooth transition.
  - After 5 days the starting point scenario is fully phased out.

In [4]:
```
datayear = 2025
scenarioyearstart = 1981
scenarioyearstop = 1996
numscen = 15 # scenarios to consider uncertainty for

phaseinoffsetdays = 2 # also simulation step and master problem length
totaldays = 7 # length of master + scenarioproblem
# master/sub - period duration - commodity (power, hydro)
mpdp = Millisecond(Hour(1)); # power/battery time resolution in masterproblem
mpdh = Millisecond(Hour(24)); # hydro time resolution in masterproblem (not used he
spdp = Millisecond(Hour(1)); # power/battery time resolution in scenarioproblems
spdh = Millisecond(Hour(24)); # hydro time resolution in scenarioproblems (not usec
timeresolutioninfo = (totaldays, phaseinoffsetdays, mpdp, mpdh, spdp, spdh);
```

```
# Simple start time
tnormal = FixedDataTwoTime(getisoyearstart(datayear),getisoyearstart(scenarioyears
# Start time that considers in-phasing of uncertainty
phaseinoffset = Millisecond(Day(phaseinoffsetdays)) # phase in straight away from
phaseindelta = Millisecond(Day(5)) # Phase in the second stage scenario over 5 days
phaseinsteps = 5 # Phase in second stage scenario in 5 steps
tphasein = PhaseinTwoTime(getisoyearstart(datayear),getisoyearstart(scenarioyearst
```

Make and load dataset

In [5]:

```
# Solar, battery and transmission parameters
transmcap = 5 # MW
transmeff = 0.995 # Small loss to avoid unnecessary transfers
storagecap = 0.01 # GWh
chargecap = 7.0 # MW
lossbattery = 0.075 # the whole loss when the battery charges
solarcap = 15.0 # MW

# Power balances for price areas and transmission
elements = getelements(JSON.parsefile("priceDMK.json"));
addexogenbalance!(elements, "PowerBalance_ExternalHub", "Power", "PriceDMK")

addbalance!(elements, "PowerBalance_HomeHub", "Power")

addpowertrans!(elements, "PowerBalance_ExternalHub", "PowerBalance_HomeHub", transm
addpowertrans!(elements, "PowerBalance_HomeHub", "PowerBalance_ExternalHub", transm

# Add battery
addbattery!(elements, "Battery", "PowerBalance_HomeHub", storagecap, lossbattery,

# Add solar production as an RHSTerm
path = "testprofiles_1981_2010.csv" # profiles from https://www.nve.no/energi/analy
dfmt = dateformat"yyyy-mm-dd HH:MM:SS"
df = CSV.read(path, DataFrame)
df.Timestamp = DateTime.(df.Timestamp, dfmt)
@assert issorted(df.Timestamp)
start = first(df.Timestamp)
numperiods = length(df.Timestamp)
push!(elements, DataElement(TIMEINDEX_CONCEPT, "RangeTimeIndex", "SolProfileTimeInd
        Dict("Start" => start, "Delta" => Hour(1), "Steps" => numperiods)))
push!(elements, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "SolProfilValue
        Dict("Vector" => df.SolarGER)))
push!(elements, DataElement(TIMEVECTOR_CONCEPT, "RotatingTimeVector", "SolProfil",
        Dict(TIMEVALUES_CONCEPT => "SolProfilValues", TIMEINDEX_CONCEPT => "SolProf
push!(elements, DataElement(PARAM_CONCEPT, "MWToGWhSeriesParam", "SolParam", Dict(
addrhsterm!(elements, "SolParam", "PowerBalance_HomeHub", DIRECTIONIN)

# Add scenariotimeperiod. Only data from weather scenarios 1981-1995 is read.
addscenariotimeperiod!(elements, scenarioyearstart, scenarioyearstop);
```

Add horizons and make master and subobjects

In [6]:

```
function singlemakestochasticobjects(elements, days, offset, pdp, pdh)

    # Add horizons to elements
    power_horizon = SequentialHorizon(ceil(Int64, Day(days)/pdp), pdp; offset)
    hydro_horizon = SequentialHorizon(ceil(Int64, Day(days)/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 and group into subsystems
    modelobjects = getmodelobjects(elements)

    return collect(values(modelobjects))
end

function singlemakemastersubobjects!(inputs)
    (elements, totaldays, numscen, scenarioyearstart, phaseinoffsetdays, timeresolu
    (totaldays, phaseinoffsetdays, mpdp, mpdh, spdp, spdh) = timeresolutioninfo

    # Make masterproblem objects
    masterobjects = singlemakestochasticobjects(copy(elements), phaseinoffsetdays,

    # Make scenarioproblem objects with offsets
    subscenarioobjects = []
    for scenario in 1:numscen
        offset = ScenarioOffset(MsTimeDelta(Day(phaseinoffsetdays)), MsTimeDelta(g
        push!(subscenarioobjects, singlemakestochasticobjects(copy(elements), tota
    end

    return (masterobjects, subscenarioobjects)
end

inputs = (elements, totaldays, numscen, scenarioyearstart, phaseinoffsetdays, time
(masterobjects, subobjects) = singlemakemastersubobjects!(inputs);
```

Initialize master and subproblems and run first iteration

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
reltol = 0.0001 # relative tolerance

shortstartstorage = 50 # start and end storage batteries in first time step (not u
medstartstorage = 65 # start and end storage hydro in first time step (not used)
medendvaluesdicts = Dict()
storageinfo = (shortstartstorage, medstartstorage, medendvaluesdicts)

(master, subs, states, cuts) = stochastic_init(masterobjects, subobjects, true, st
```

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

In [8]:
```
startstates_ = getstatevariables(master.objects)
getoutgoingstates!(master, startstates_)
startstates = Dict()
for var in keys(startstates_)
    startstates[getinstancename(first(getvarout(var)))] = startstates_[var]
end
```

Collect results from masterproblem

In [9]:
```
masterobjects = Dict(zip([getid(obj) for obj in master.objects],master.objects))
resultobjects = master.objects # collect results for all areas
```

```
prices, rhstermvalues, production, consumption, hydrolevels, batterylevels, powerba
```

Simulate next time steps

- Simulate next time steps, store results and plot results

In [10]:
```
function singlestochastic!(master, subs, states, cuts, startstates, medendvaluesdi

    # Init cutparameters
    cutparameters = Vector{Tuple{Float64, Dict{StateVariableInfo, Float64}}}(undef

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

    update!(master, t)

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

        substorages = getstorages(getobjects(sub))

        if short
            setendstates!(sub, substorages, startstates) # set end reservoir
#         else
#             subendvaluesobj = sub.objects[findfirst(x -> getid(x) == Id(BOUNDARY(
#             subendvalues = [medendvaluesdicts[i][getinstancename(getid(obj))] for
#             updateendvalues!(sub, subendvaluesobj, subendvalues)
        end

        update!(sub, t) # update parameters given problem start time of scenario
    end

    lb = cuts.lower_bound
    ub = 0
    cutreuse = true # reuse cut from last time step
    iterate_convergence!(master, subs, cuts, cutparameters, states, numscen, cutreu
end;
```

In [11]:
```
# steps = 16;
steps = (getisoyearstart(scenarioyearstop) - getisoyearstart(scenarioyearstart)).va
```

In [12]:
```
step = 2
tnormal += Day(2) # jump to next time step
tphasein = PhaseinTwoTime(getdatatime(tnormal), getscenariotime(tnormal), getscenar
# display(tnormal)

totaltime = @elapsed while step <= steps

    # Stochastic sub system
    singlestochastic!(master, subs, states, cuts, startstates, medendvaluesdicts, t

    # Update startstates
    startstates_ = getstatevariables(master.objects)
    getoutgoingstates!(master, startstates_)
    for var in keys(startstates_)
        # value = round(startstates_[var], digits=10) # avoid approx 0 negative val
        startstates[getinstancename(first(getvarout(var)))] = startstates_[var]
    end

    # Collect results
```

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

    # Jump to next time step
    step += 1
    tnormal += Day(2)
    tphasein = PhaseinTwoTime(getdatatime(tnormal), getscenariotime(tnormal), gets
#     display(tnormal)
end

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

```
"The simulation took: 1.4951274266666668 minutes"
"Time usage per timestep: 0.032734043276774316 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

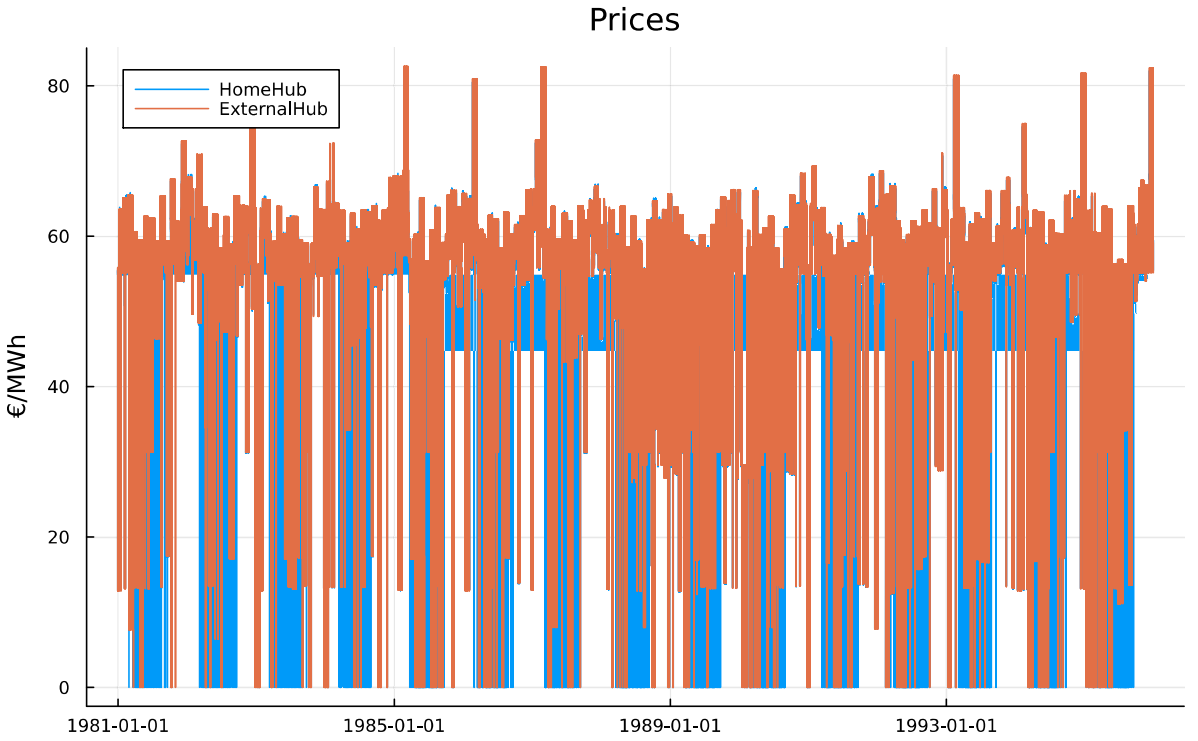
In [13]:
```
# Only keep rhsterms that have at least one value (TODO: Do the same for sypply and
rhstermtotals = dropdims(sum(rhstermvalues,dims=1),dims=1)
rhstermsupplyidx = []
rhstermdemandidx = []

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_")[

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

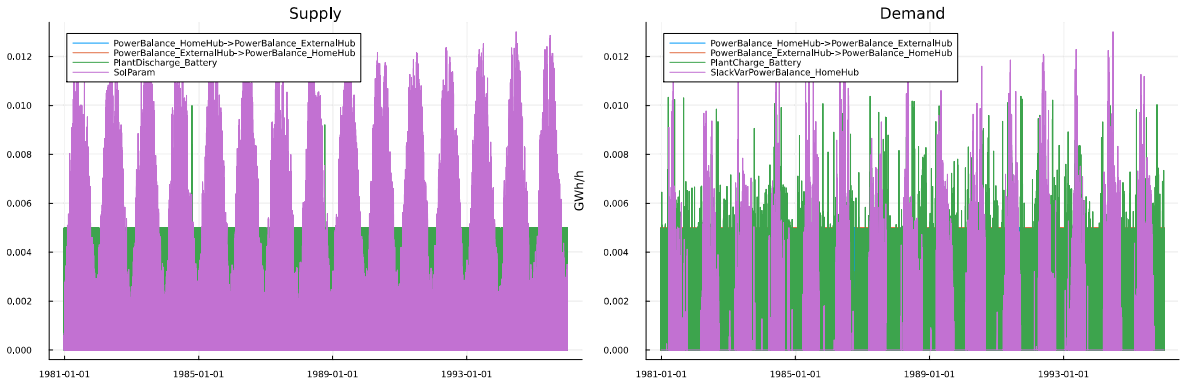
In [14]:
```
# folder =
# areaprices = rename!(DataFrame(prices, :auto),powerbalancenames)
```

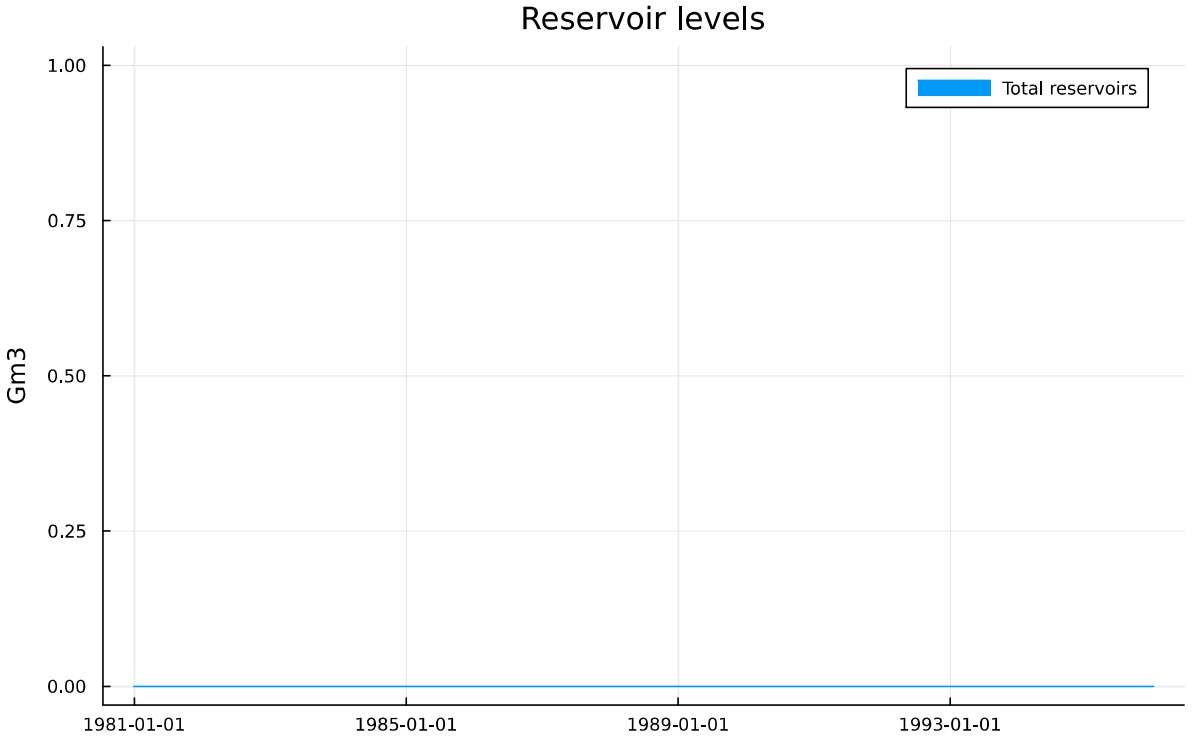
```
# areaprices[!,:time] = x1
# CSV.write(folder * "prices.csv", areaprices)

# 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);
```

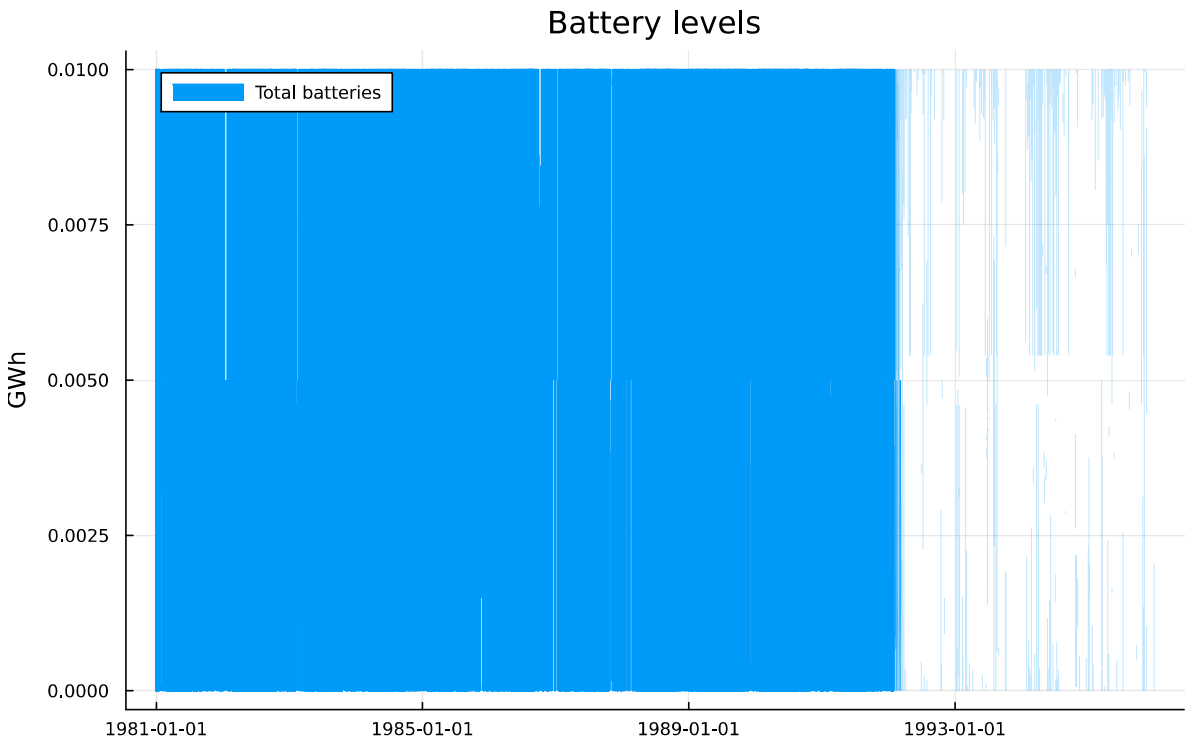
In [15]:

```
# Plot prices
display(plot(x1, prices/1000, labels=reshape(powerbalancenames,1,length(powerbalan

# # Plot supplies and demands
supplychart = plot(x1, supplyvalues,labels=reshape(supplynames,1,length(supplyname
demandchart = plot(x1, demandvalues,labels=reshape(demandnames,1,length(demandname
display(plot([supplychart,demandchart]...,layout=(1,2),size=(1600,500)))
# supplychart = areaplot(x1, sum(supplyvalues,dims=2),title="Supply", ylabel = "GW
# demandchart = areaplot(x1, sum(demandvalues,dims=2),title="Demand", ylabel = "GW
# display(plot([supplychart,demandchart]...,layout=(1,2),size=(900,500)))

# Plot storages
# display(areaplot(x2, hydrolevels,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)

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


Prices
HomeHub
ExternalHub
€/MWh
80
60
40
20
0
1981-01-01
1985-01-01
1989-01-01
1993-01-01


Supply
PowerBalance_HomeHub->PowerBalance_ExternalHub
PowerBalance_ExternalHub->PowerBalance_HomeHub
PlantDischarge_Battery
SolParam
GWh/h
Demand
PowerBalance_HomeHub->PowerBalance_ExternalHub
PowerBalance_ExternalHub->PowerBalance_HomeHub
PlantCharge_Battery
SlackVarPowerBalance_HomeHub


Reservoir levels
Total reservoirs
Gm3
1.00
0.75
0.50
0.25
0.00
1981-01-01
1985-01-01
1989-01-01
1993-01-01



4×4 DataFrame

Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
	String?	Float64?	String?	Float64?
1	SolParam	0.0146608	PowerBalance_HomeHub->PowerBalan…	0.0139459
2	PowerBalance_HomeHub->PowerBalan…	0.0138761	PlantCharge_Battery	0.00328465
3	PlantDischarge_Battery	0.00303864	SlackVarPowerBalance_HomeHub	0.00156142
4	PowerBalance_ExternalHub->PowerB…	0.00109255	PowerBalance_ExternalHub->PowerB…	0.00109804

1×2 DataFrame

Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing
	Float64	Float64
1	0.0326681	0.01989

In [ ]:

In [ ]: