In [1]:
```
using DataFrames, Plots, Statistics, JSON, Clp
plotlyjs()
include(joinpath(dirname(pwd()),"src/TuLiPa.jl")); # using Dates, JuMP, HiGHS, CSV, Clustering
include("Demo 1a - Deterministic power market with dummy data.jl"); # Hid some functions here to make the notebook more readable
```

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

```
┌ Warning: backend `PlotlyJS` is not installed.
└ @ Plots C:\Users\jgrc\.julia\packages\Plots\bMtsB\src\backends.jl:43
```

Demo 1a - Deterministic power market with dummy data

We show how you can set up a simple deterministic power market problem with TuLiPa. The problem will minimize the cost of covering the demand with production from different technologies.

The problem contains:

- Two price areas: NO2 and GER (€/GWh)
- Transmission between the areas
- Inelastic demand in both areas
- Wind and solar in both areas
- Two hydropower plants in NO2 (one reservoir and one run-of-river)
- Four thermal plants in GER (gas, coal, biogas and nuclear)
- Two commodities: Power (GWh and MW) and Hydro (Mm3 and m3/s)
- Two horizons / time-resolutions: The market balance (power commodity) with daily resolution. The waterbalance (hydro commodity) is weekly. The assumption here is that the waterbalances do not need a very fine time-resolution, so we take advantage of TuLiPa supporting different horizons in the same problem. This gives a smaller problem with less balances and variables.
- Different input data for different scenarios: Constants or time-series data for different weather scenarios and model years.

These elements need their own model objects. Each model object (e.g. hydropower production or the water balance) has an id, horizon (with time resolution) and special properties. The properties of the object contain information about which elements must be built and updated in the LP problem, as well as a dataset. The dataset can contain values for different weather scenarios, and different model years (e.g. installed capacity in 2021 or 2025). When we update the problem with a weather scenario (scenario time) and a model year (data time), the model objects will build the LP problem for this input data. Then the problem can be solved.

In this demo we use dummy data, and profiles for wind, solar, demand and inflow from https://www.nve.no/energi/analyser-og-statistikk/vaerdatasett-for-kraftsystemmodellene/

Overview of the document:

1 - Make a dataset

2 - Build, update and solve a problem

3 - Run several scenarios and look at results

1 - Make dataset

DataElements

The model objects are assembled from a list of dataelements. The dataelements have a concept, concepttype, unique instancename and a data field. The conceptname and concepttype decide how the dataelement should be converted to a model object, while the data field holds all the data that should be put into the model object.

The data elements are stored in a flat structure where each data element holds its information and possibly references to other dataelements they are linked to (for example can an upper capacity hold references to its parameter (stored in another data element), and references to the variable this upper capacity should apply to).

Helping functions to define a dataset of DataElements

In [2]:
```
# We have hid these functions in the file Demo 1a - Deterministic power market with dummy data.jl
# Alternatively take a look at Demo 1b where all the code is included in the notebook
elements = gettestdataset();
```

Take a look at the dataelements

In [3]:
```
println(string("Number of elements: ", length(elements)))
printdicts(elements, 10) # print first 10 elements
# printdicts(elements) # print all elements
```

```
Number of elements: 108
[
  {
    "conceptname": "Balance",
    "typename": "BaseBalance",
    "instancename": "PowerBalance_NO2",
    "value": {
      "Commodity": "Power"
    }
  },
  {
    "conceptname": "Flow",
    "typename": "BaseFlow",
    "instancename": "SlackVarPowerBalance_NO2",
    "value": {}
  },
  {
    "conceptname": "Arrow",
    "typename": "BaseArrow",
    "instancename": "SlackArrowPowerBalance_NO2",
    "value": {
      "Balance": "PowerBalance_NO2",
      "Flow": "SlackVarPowerBalance_NO2",
      "Conversion": 1.0,
      "Direction": "Out"
    }
  },
  {
    "conceptname": "Balance",
    "typename": "BaseBalance",
    "instancename": "PowerBalance_GER",
    "value": {
      "Commodity": "Power"
    }
  },
  {
    "conceptname": "Flow",
    "typename": "BaseFlow",
    "instancename": "SlackVarPowerBalance_GER",
    "value": {}
  },
  {
    "conceptname": "Arrow",
    "typename": "BaseArrow",
    "instancename": "SlackArrowPowerBalance_GER",
    "value": {
      "Balance": "PowerBalance_GER",
      "Flow": "SlackVarPowerBalance_GER",
      "Conversion": 1.0,
      "Direction": "Out"
    }
  },
  {
    "conceptname": "Flow",
    "typename": "BaseFlow",
    "instancename": "PowerBalance_NO2->PowerBalance_GER",
    "value": {}
  },
  {
    "conceptname": "Arrow",
    "typename": "BaseArrow",
    "instancename": "PowerBalance_NO2->PowerBalance_GERFrom",
    "value": {
      "Balance": "PowerBalance_NO2",
      "Flow": "PowerBalance_NO2->PowerBalance_GER",
      "Conversion": 1.0,
      "Direction": "Out"
    }
  },
  {
    "conceptname": "Arrow",
    "typename": "BaseArrow",
    "instancename": "PowerBalance_NO2->PowerBalance_GERTo",
    "value": {
      "Balance": "PowerBalance_GER",
      "Flow": "PowerBalance_NO2->PowerBalance_GER",
      "Conversion": 0.97,
      "Direction": "In"
    }
  },
  {
    "conceptname": "Capacity",
    "typename": "PositiveCapacity",
    "instancename": "CapPowerBalance_NO2->PowerBalance_GER",
    "value": {
      "Param": "CapPowerBalance_NO2->PowerBalance_GER",
      "WhichInstance": "PowerBalance_NO2->PowerBalance_GER",
      "WhichConcept": "Flow",
      "Bound": "Upper"
    }
  }
]
```


Runsettings

Define some model settings that completes the dataset.

In [4]:
```
# Select the weather scenarios 1981-1982 (some of the profiles are 1981-2010, we only want to use two years)
# Used by RotatingTimeVector to reduce the amount of data
scenarioyearstart = 1981
scenarioyearstop = 1983
addscenariotimeperiod!(elements, "ScenarioTimePeriod", getisoyearstart(scenarioyearstart), getisoyearstart(scenarioyearstop));
```

In [5]:
```
# We choose the horizon (time-resolution) of the commodities. We set the duration of the horizons to 3 years
# We want the variables connected to power (daily) to be more detailed than the hydro variables (weekly)
power_horizon = SequentialHorizon(364*3, Day(1))
hydro_horizon = SequentialHorizon(52*3, Week(1))
set_horizon!(elements, "Power", power_horizon)
set_horizon!(elements, "Hydro", hydro_horizon);
```

In [6]:
```
# Storages have state-dependant variables that need a boundary condition
# We set the starting storage to be equal to the ending storage, x[0] = x[T] (for horizon where t in 1:T)
push!(elements, getelement(BOUNDARYCONDITION_CONCEPT, "StartEqualStop", "StartEqualStop_StorageResNO2",
        (WHICHINSTANCE, "StorageResNO2"),
        (WHICHCONCEPT, STORAGE_CONCEPT)));
```

Make model objects from dataelements

In [7]:
```
modelobjects = getmodelobjects(elements)

keys(modelobjects)
```

Out[7]:
```
KeySet for a Dict{Any, Any} with 15 entries. Keys:
  Id("Flow", "SlackVarPowerBalance_NO2")
  Id("Flow", "GenCoalGER")
  Id("Balance", "PowerBalance_GER")
  Id("Flow", "ReleaseResNO2")
  Id("Flow", "PowerBalance_GER->PowerBalance_NO2")
  Id("Balance", "ResNO2")
  Id("Flow", "GenBioGER")
  Id("Flow", "GenNucGER")
  Id("Balance", "PowerBalance_NO2")
  Id("Flow", "SpillResNO2")
  Id("Flow", "SlackVarPowerBalance_GER")
  Id("Flow", "GenGasGER")
  Id("BoundaryCondition", "StartEqualStop_StorageResNO2")
  Id("Storage", "StorageResNO2")
  Id("Flow", "PowerBalance_NO2->PowerBalance_GER")
```

Take a look at the hydropower release variable

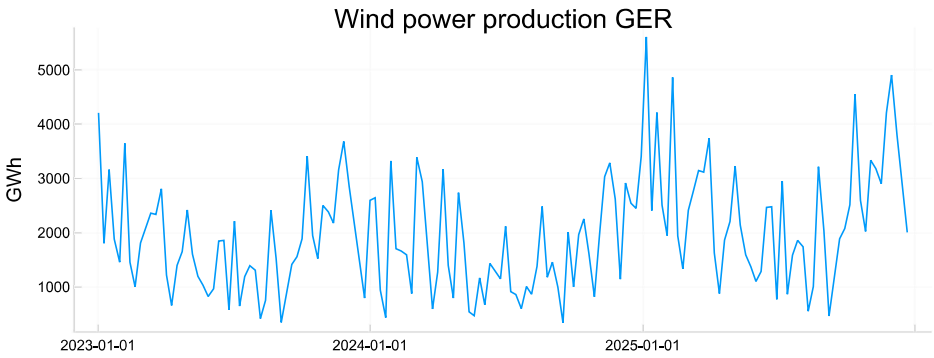
In [8]:
```
println("We display the fields in the struct")
display(fieldnames(typeof(modelobjects[Id("Flow", "ReleaseResNO2")])))

println("\nThe Id shows that the struct is of type Flow")
display(getid(modelobjects[Id("Flow", "ReleaseResNO2")]))

println("\nThe Horizon consist of 1092 periods with timedelta of 1 day (total 3 years)")
display(gethorizon(modelobjects[Id("Flow", "ReleaseResNO2")]))

println("\nThe lower bound is LowerZeroCapacity so that the variable is non-negative")
display(getlb(modelobjects[Id("Flow", "ReleaseResNO2")]))

println("\nThe upper bound is 6000 m3/s, which is stored in a M3SToMM3SeriesParam. M3SToMM3SeriesParam converts the upper bound into
display(getub(modelobjects[Id("Flow", "ReleaseResNO2")]))

println("\nThere are two arrows. The first says that the release takes water out from the water balance and the second says that it c
for arrow in getarrows(modelobjects[Id("Flow", "ReleaseResNO2")])
    display(getid(arrow))
#     display(arrow)
end
```

```
We display the fields in the struct
(:id, :horizon, :ub, :lb, :costs, :sumcost, :arrows, :metadata)
The Id shows that the struct is of type Flow
Id("Flow", "ReleaseResNO2")
The Horizon consist of 1092 periods with timedelta of 1 day (total 3 years)
SequentialHorizon(SequentialPeriods([(1092, Millisecond(86400000))]), nothing)
The lower bound is LowerZeroCapacity so that the variable is non-negative
LowerZeroCapacity()
The upper bound is 6000 m3/s, which is stored in a M3SToMM3SeriesParam. M3SToMM3SeriesParam converts the upper bound into Mm3 depend
ing on the horizon of the problem
PositiveCapacity(Id("Capacity", "CapReleaseResNO2"), M3SToMM3SeriesParam{ConstantTimeVector, ConstantTimeVector}(ConstantTimeVector
(6000.0), ConstantTimeVector(1.0)), true)
There are two arrows. The first says that the release takes water out from the water balance and the second says that it contributes
into the power balance
Id("Arrow", "WaterDemandReleaseResNO2")

Id("Arrow", "PowerSupplyReleaseResNO2")
```

Take a look at the wind parameter for GER

In [9]:
```
# It is stored in one of the rhsterms of PowerBalance_NO2
rhsterm = getrhsterms(modelobjects[Id("Balance", "PowerBalance_GER")])[1]

# We start with the installed capacity in 2023 and the weather scenario 1981
start = TwoTime(getisoyearstart(2023), getisoyearstart(1981))
```

```
# We look at the weekly production in GWh for three years from the starting point
horizon = SequentialHorizon(52*3, Week(1))

# Vectors to store results
i = Vector{typeof(getdatatime(start))}(undef, getnumperiods(horizon))
v = zeros(getnumperiods(horizon))

# Query results for each period in the horizon
for t in 1:getnumperiods(horizon)
    querystart = getstarttime(horizon, t, start)
    querydelta = gettimedelta(horizon, t)
    i[t] = getdatatime(querystart)
    v[t] = getparamvalue(rhsterm, querystart, querydelta)
end

# Plot
display(plot(i, v, legend=false, size=(700,300), ylabel = "GWh", title = "Wind power production GER"))
```



- In 1.3 we define the installed wind capacity for 2021 and 2025. We therefore see that the wind power production increases in 2025. This is because the level is an InfiniteTimeVector.
- Since we chose in 1.5 to only use the weather years 1981 and 1982, the profile reuses 1981 for the third year. This is because the profile is a RotatingTimeVector

2 - Show how a problem can be built, updated and solved

2.1 - Initialize a JuMP_Prob problem

We build a problem object for a minimization problem. The problem object consist of:

- A JuMP model with the solver CLP
  - JuMP is a modelling framework for working with optimization problems: https://github.com/jump-dev/JuMP.jl
  - Clp is an open-source LP-solver: https://github.com/jump-dev/Clp.jl
- The list of model objects

The problem is also initialized. Variables and balances are built, and variables are put into balances and the objective function if coefficients are constant regardless of scenario or horizon period.

In [10]:
```
mymodel = JuMP.Model(Clp.Optimizer)
set_silent(mymodel)

prob = JuMP_Prob(modelobjects, mymodel)
# prob = HiGHS_Prob(modelobjects) # we could also have used HiGHS_Prob, but printing the mathematical model is not supported there

prob.model
```

Out[10]:
```
A JuMP Model
Minimization problem with:
Variables: 10141
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.EqualTo{Float64}`: 2341 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 10140 constraints
`VariableRef`-in-`MathOptInterface.LessThan{Float64}`: 7800 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: Clp
Names registered in the model: BalancePowerBalance_GER, BalancePowerBalance_NO2, BalanceResNO2, BoundaryConditionEqStartEqualStop_St
orageResNO2, FlowGenBioGER, FlowGenCoalGER, FlowGenGasGER, FlowGenNucGER, FlowPowerBalance_GER->PowerBalance_NO2, FlowPowerBalance_N
O2->PowerBalance_GER, FlowReleaseResNO2, FlowSlackVarPowerBalance_GER, FlowSlackVarPowerBalance_NO2, FlowSpillResNO2, StorageStartSt
orageResNO2, StorageStorageResNO2
```

The hydrorbalance has been initialized. But the inflow is not added since the value depends on the chosen scenario and period in the horizon.

In [11]:
```
display(prob.model[:BalanceResNO2][1])
```

BalanceResNO2[1] :

$$FlowReleaseResNO2_1 + FlowReleaseResNO2_2 + FlowReleaseResNO2_3 + FlowReleaseResNO2_4 + FlowReleaseResNO2_5 + FlowRele$$
$$- StorageStartStorageResNO2_1 = 0.0$$

2.2 - Choose a starting point for the horizon and update the problem

Variables are now put into balances and the objective function if coefficients are dependant on the scenario or horizon period. RHSTerms like the inflow are also updated, but it is stored in the problem object rather than updated in the JuMP model straight away. When solve! is called, the problem object will check if any of the RHSTerm has been updated. If that is the case, the JuMP model will be updated with the RHSTerms.

In [12]:
```
t = TwoTime(getisoyearstart(2021), getisoyearstart(1981))

update!(prob, t)
```

2.3 - Solve the problem and display the solution

In [13]:
```
@time solve!(prob)

getobjectivevalue(prob)
```

```
  5.450143 seconds (9.89 M allocations: 553.758 MiB, 5.17% gc time, 98.80% compilation time)
```

Out[13]:
```
9.61646303403377e10
```

Now the hydropower balance also includes the inflow, which is 156 Mm3 in the second week of the problem.

In [14]:
```
display(prob.model[:BalanceResNO2][1])
```

BalanceResNO2[1] :

$$FlowReleaseResNO2_1 + FlowReleaseResNO2_2 + FlowReleaseResNO2_3 + FlowReleaseResNO2_4 + FlowReleaseResNO2_5 + FlowRele$$
$$- StorageStartStorageResNO2_1 = 311.2851168$$

2.4 - Update the problem with a start time from another model year and weather scenario, and solve the problem again

- We get a new solution to the problem
- The inflow is now different since we start at the weather scenario 1982 and the inflow level from 2025.

In [15]:
```
t = TwoTime(getisoyearstart(2024), getisoyearstart(1982))

update!(prob, t)

solve!(prob)

display(getobjectivevalue(prob))

display(prob.model[:BalanceResNO2][1])
```

```
8.194441022906743e10
```

BalanceResNO2[1] :

$$FlowReleaseResNO2_1 + FlowReleaseResNO2_2 + FlowReleaseResNO2_3 + FlowReleaseResNO2_4 + FlowReleaseResNO2_5 + FlowRele$$
$$- StorageStartStorageResNO2_1 = 53.7303888$$

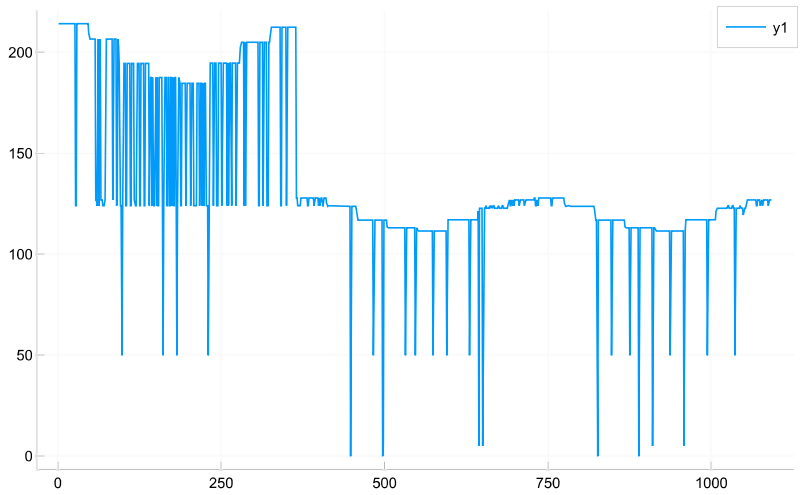
2.5 - Look at prices

Collect daily prices from the model using the JuMP result framework

The prices are in EUR/MWh. The price decline from 2025 is a result of the new gas price level in 2025.

In [16]:
```
plot(-dual.(prob.model[:BalancePowerBalance_GER])/1000)
# plot(value.(prob.model[:FlowGenGasGER])) # Or we could get the daily gas production in GWh
```

Out[16]:

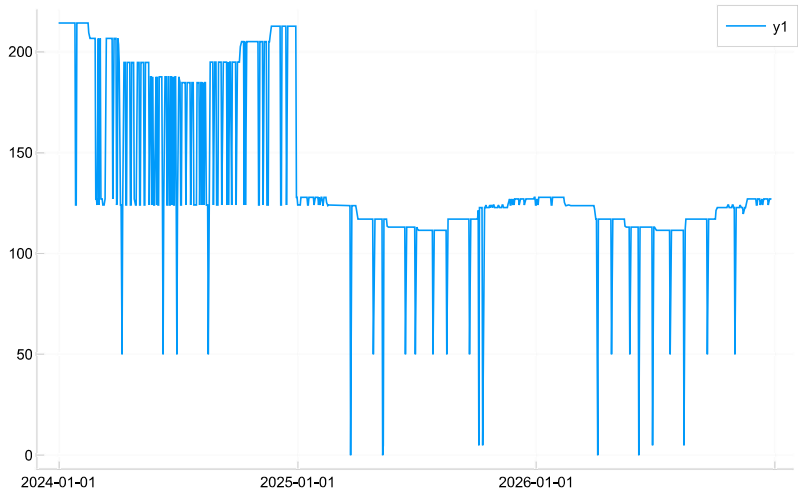


Or we could use the result functions built into JuMP_Prob

- getcondual() to get prices
- getvarvalue()
- getrhsterm()

In [17]:
```
numperiods = getnumperiods(power_horizon)
x = [getdatatime(t) + getstartduration(power_horizon, s) for s in 1:numperiods]
y = [-getcondual(prob, Id("Balance","PowerBalance_GER"), s) for s in 1:numperiods]/1000
plot(x,y)
```

Out[17]:



3 - Run several scenarios and look at more results

3.1 Function that builds, updates and solves the problem, and plot the results

The function runs several scenarios and plot prices, supplies, demands and hydro storages for each scenario. The function only plots results for chosen price areas.

In Demo 2 a more complex version of this function is used, which support more types of objects. This includes AdaptiveHorizons (horizon built based on residual load), Horizons with offset, SegmentedArrow (for hydropower plants with PQ-curves), BaseAggSupplyCurve (for aggregated power plants) and ExogenBalance (exogen price areas)

In [18]:
```
# We have hid this function in the julia file: Demo 1a - Deterministic power market with dummy data.jl
```

3.2 - Run scenarios

We choose scenarios that start in the model year 2024 and weather scenarios that start in 1981/1982/1983

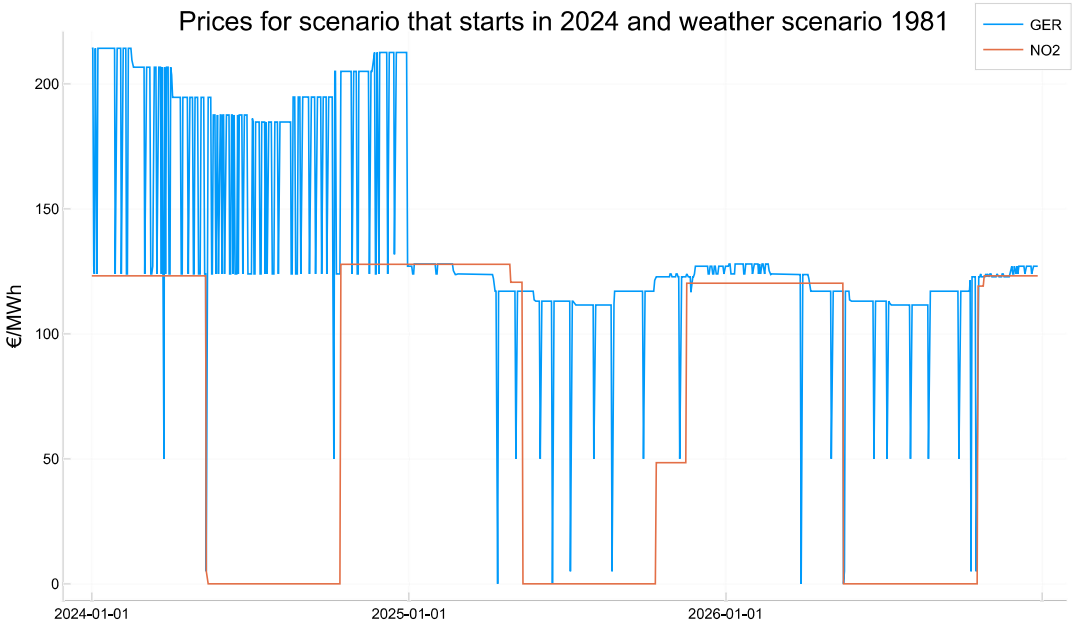
We get expected results for the deterministic market problem

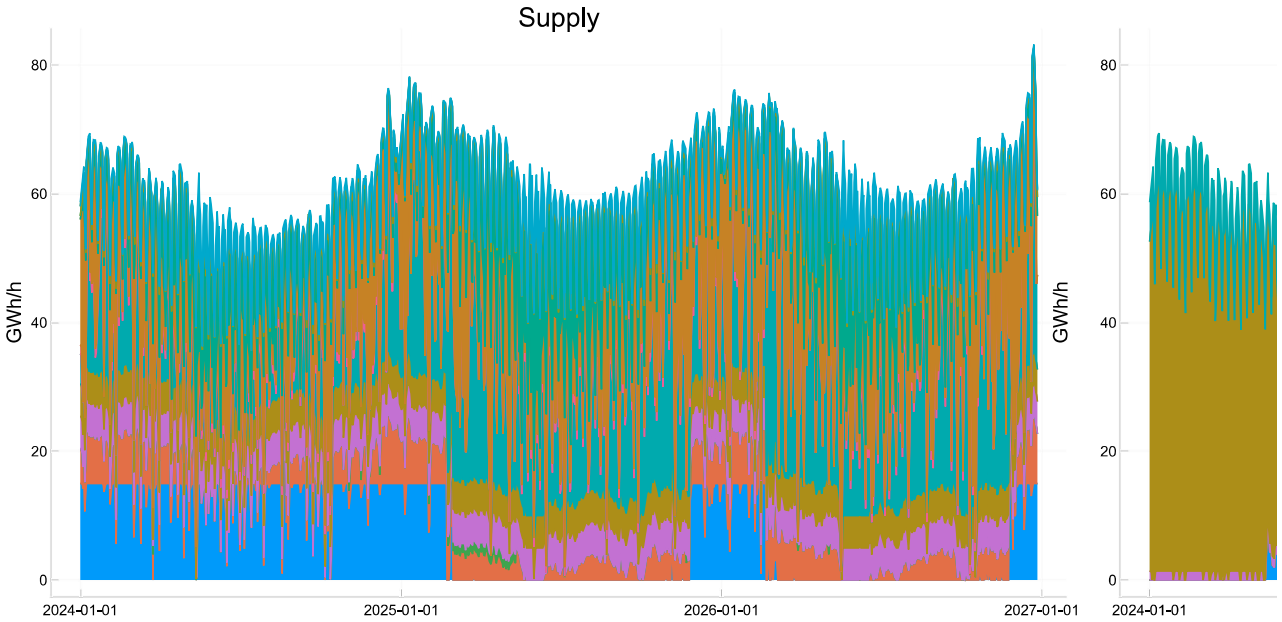
- The supply equals the demand at all times. When the RoR production is very high during the spring flood, the slack variable in the power balance is used as a demand.
- The different scenarios gives prices decided by the marginal cost of the most expensive production technology at any time.
- The hydro storage is emptied before the spring flood and stores water for the winter

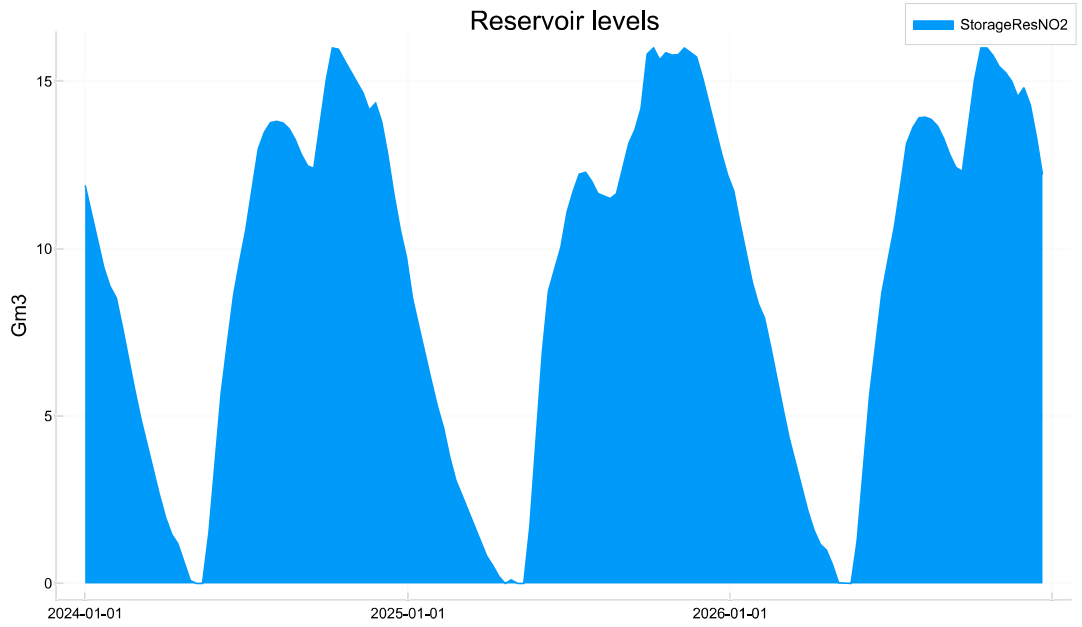
- Due to spillage during the spring flood and the model being deterministic, the power price in NO2 is very flat and goes to zero during the summer.

In [19]:
```
scenarios = getscenarios(2024; years=1981:1983)

runscenarios(scenarios, modelobjects)

# alternative for only seeing results from one price area
# resultobjects = getpowerobjects(modelobjects,["NO2"])
# runscenarios(scenarios, modelobjects, resultobjects)
```

```
Objective value in scenario 1: 8.264928001510184e10
Objective value in scenario 2: 8.19444102290675e10
Objective value in scenario 3: 8.303707657019687e10
```





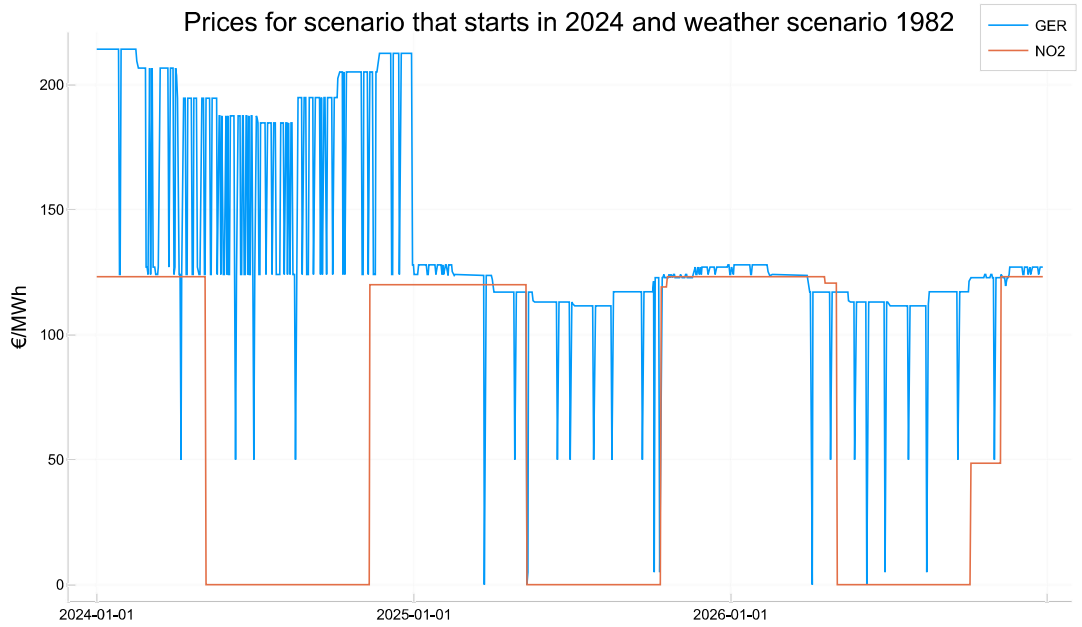


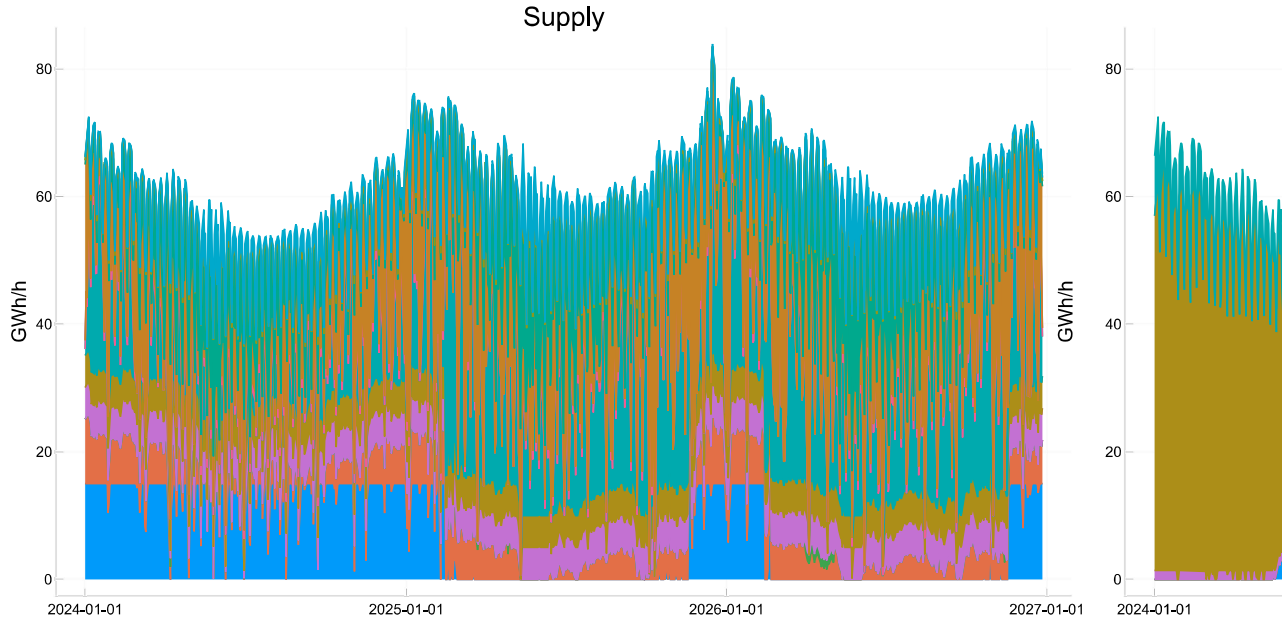
11×4 DataFrame

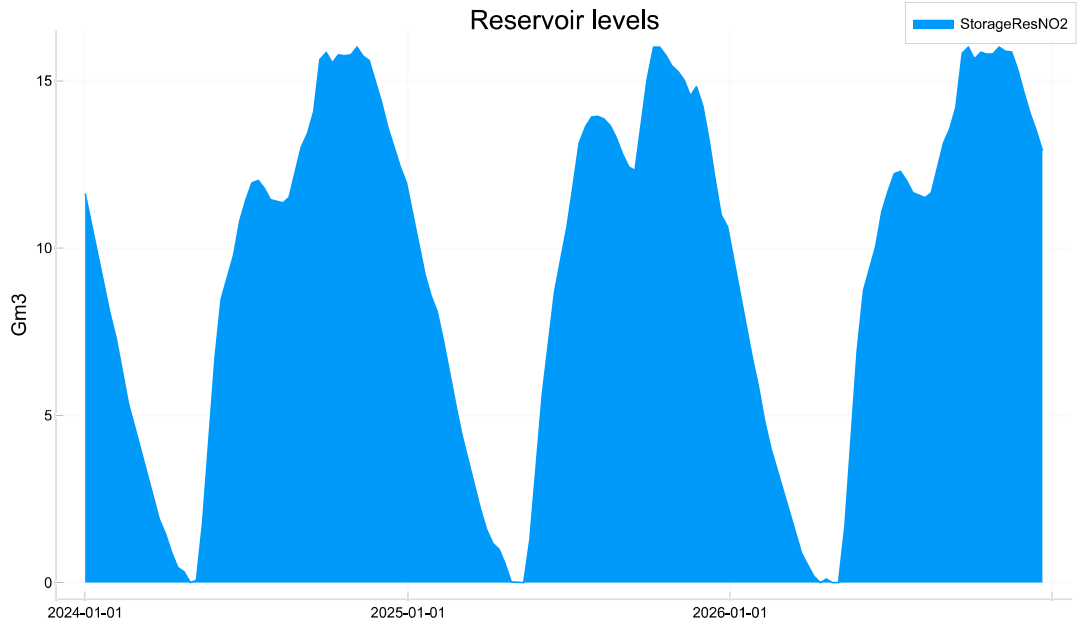
Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
	String?	Float64?	String?	Float64?
1	GenGasGER	131.058	DemandGER	468.054
2	WindGER	111.944	DemandNO2	48.4256
3	SolarGER	70.6705	PowerBalance_NO2->PowerBalance_G…	8.62053
4	GenCoalGER	60.4118	SlackVarPowerBalance_NO2	1.80325
5	GenNucGER	43.5204	PowerBalance_GER->PowerBalance_N…	0.984684
6	GenBioGER	43.1119	SlackVarPowerBalance_GER	0.0400589
7	ReleaseResNO2	36.954	missing	missing
8	RoRNO2	15.9198	missing	missing
9	PowerBalance_NO2->PowerBalance_G…	8.36191	missing	missing
10	WindNO2	5.02039	missing	missing
11	PowerBalance_GER->PowerBalance_N…	0.955143	missing	missing

1×2 DataFrame

Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing
	Float64	Float64
1	527.929	527.929





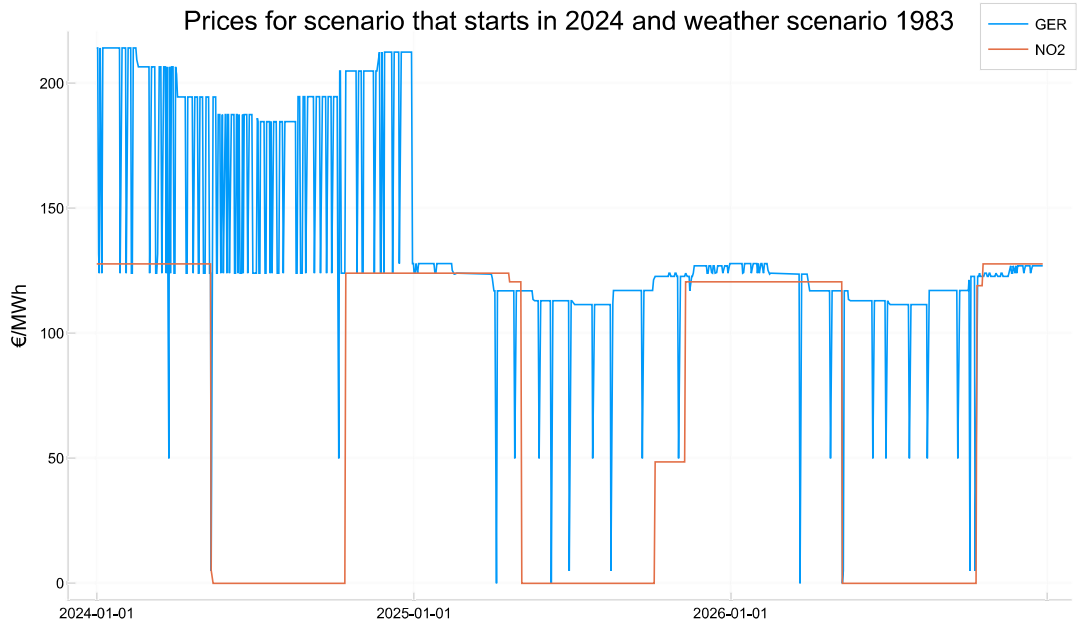


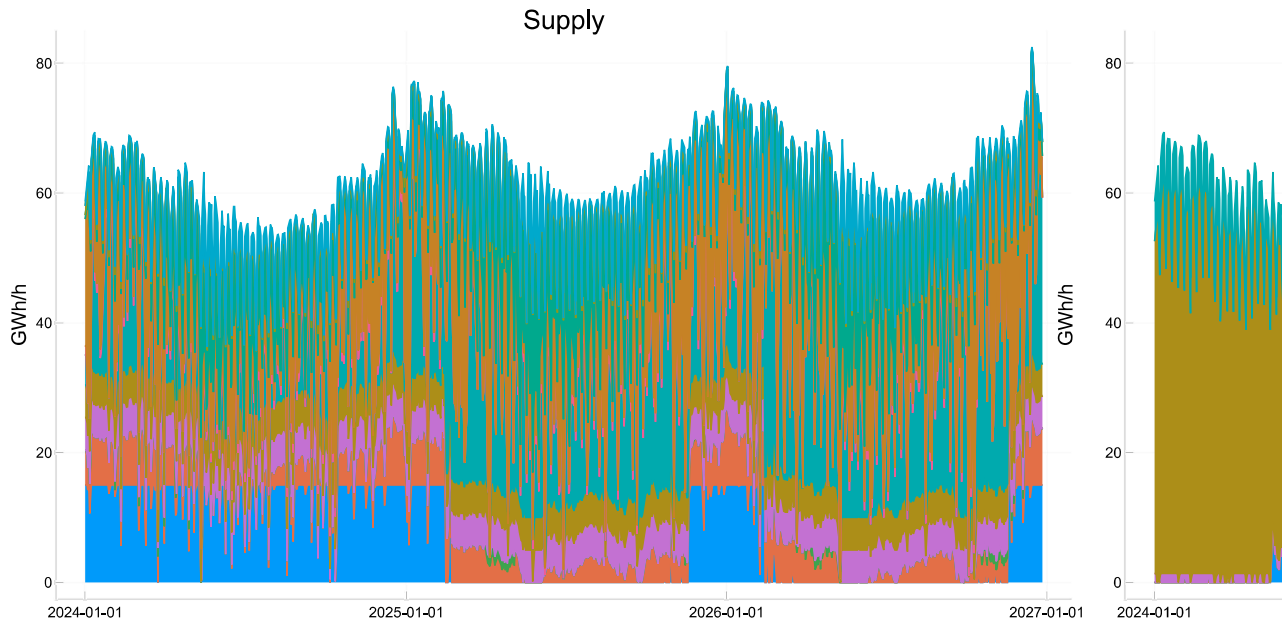
11×4 DataFrame

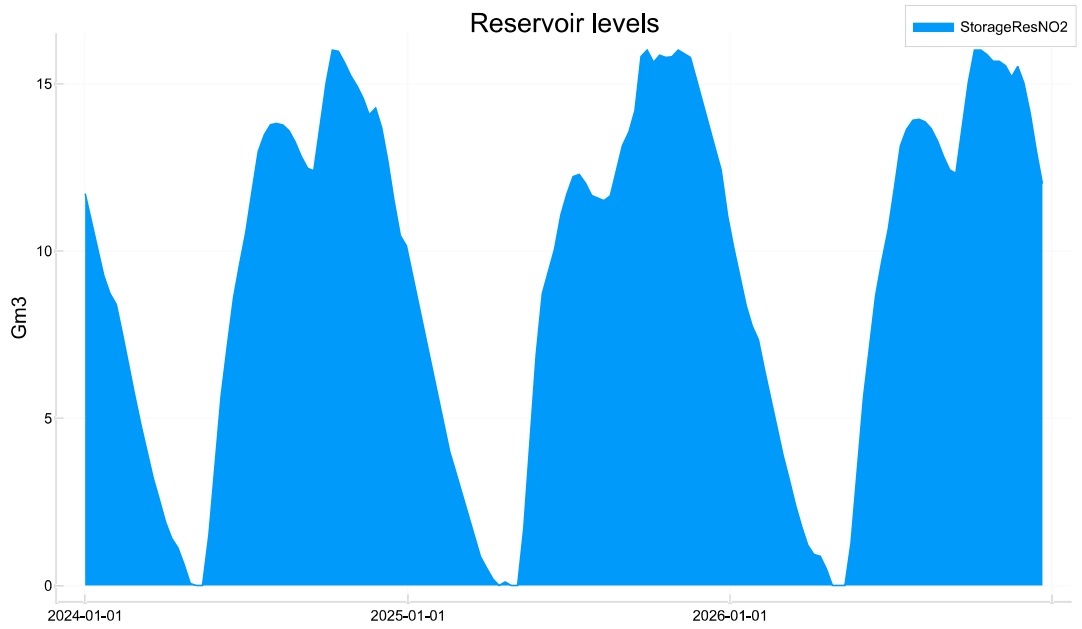
Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
	String?	Float64?	String?	Float64?
1	GenGasGER	129.634	DemandGER	467.293
2	WindGER	109.393	DemandNO2	47.626
3	SolarGER	73.102	PowerBalance_NO2->PowerBalance_G…	9.10136
4	GenCoalGER	60.1686	SlackVarPowerBalance_NO2	1.48872
5	GenNucGER	43.5209	PowerBalance_GER->PowerBalance_N…	0.412345
6	GenBioGER	43.1053	SlackVarPowerBalance_GER	0.0463198
7	ReleaseResNO2	37.2763	missing	missing
8	RoRNO2	15.4017	missing	missing
9	PowerBalance_NO2->PowerBalance_G…	8.82832	missing	missing
10	WindNO2	5.13809	missing	missing
11	PowerBalance_GER->PowerBalance_N…	0.399974	missing	missing

1×2 DataFrame

Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing
	Float64	Float64
1	525.968	525.968


Prices for scenario that starts in 2024 and weather scenario 1983
GER
NO2
€/MWh
200
150
100
50
0
2024-01-01
2025-01-01
2026-01-01


Supply
GWh/h
80
60
40
20
0
2024-01-01
2025-01-01
2026-01-01
2027-01-01
GWh/h
80
60
40
20
0
2024-01-01


Reservoir levels
StorageResNO2
Gm3
15
10
5
0
2024-01-01
2025-01-01
2026-01-01

11×4 DataFrame

Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
	String?	Float64?	String?	Float64?
1	GenGasGER	131.754	DemandGER	468.242
2	WindGER	111.142	DemandNO2	48.4851
3	SolarGER	70.6877	PowerBalance_NO2->PowerBalance_G…	8.56625
4	GenCoalGER	60.7569	SlackVarPowerBalance_NO2	1.79717
5	GenNucGER	43.5204	PowerBalance_GER->PowerBalance_N…	0.994443
6	GenBioGER	43.1119	SlackVarPowerBalance_GER	0.0463198
7	ReleaseResNO2	36.9577	missing	missing
8	RoRNO2	15.9212	missing	missing
9	PowerBalance_NO2->PowerBalance_G…	8.30926	missing	missing
10	WindNO2	5.005	missing	missing
11	PowerBalance_GER->PowerBalance_N…	0.96461	missing	missing

1×2 DataFrame

Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing
	Float64	Float64
1	528.131	528.131

In [ ]: