

```
In [92]: using DataFrames, Plots, Statistics, JSON, Clp
plotlyjs()
include(joinpath(dirname(pwd()),"src/TuLiPa.jl")); # using Dates, JuMP, HiGHS, CSV, Clustering
```

## Demo 1- 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 a dataset

### 1.1 - 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).

### 1.2 - Helping functions to define a dataset of DataElements

```
In [93]: # Read DataElement from a tuple
function getelement(concept, concrete, instance, pairs...)
    d = Dict()
    for (k, v) in pairs
        d[k] = v
    end
    DataElement(concept, concrete, instance, d)
end

# Power markets or water balances are represented with a Balance equation
# - They have a commodity which will decide the horizon (time-resolution) of the Balance
function addbalance!(elements, name, commodity)
    push!(elements, getelement(BALANCE_CONCEPT, "BaseBalance", name,
    (COMMODITY_CONCEPT, commodity)))
    # Power Balances needs a slack variable if inelastic wind, solar, or run-of-river is higher than the inelastic demand
    if commodity == "Power"
        slackname = "SlackVar" * name
        # Flows are variables that contribute into Balance equations
        push!(elements, getelement(FLOW_CONCEPT, "BaseFlow", slackname))
    end
end
```

```

slackarrowname = "SlackArrow" * name
# Arrows connect Flows and Balances
push!(elements, getelement(ARROW_CONCEPT, "BaseArrow", slackarrowname,
    (BALANCE_CONCEPT, name),
    (FLOW_CONCEPT, slackname),
    (CONVERSION_CONCEPT, 1.0), # Factor to convert Flow into the same commodity as Balance (here Power to Power, so 1)
    (DIRECTIONKEY, DIRECTIONOUT))) # Positive or negative contribution
end
end

# Rhs terms contribute to the right hand side of a Balance equation
function addrhsrterm!(elements, name, balance, direction)
    d = getelement(RHSTERM_CONCEPT, "BaseRHSTerm", name,
        (BALANCE_CONCEPT, balance),
        (PARAM_CONCEPT, name), # constant or time-series data
        (DIRECTIONKEY, direction)) # positive or negative contriution to the balance
    push!(elements, d)
end

# DataElements for a Thermal power plant
function addrhsthermal!(elements, name, balance;
    cost::Union{Real, Nothing}=nothing, # €/GWh
    cap::Union{Real, Nothing}=nothing) # MW
    genname = "Gen" * name
    costname = "MC" * name
    capname = "Cap" * name

    powerarrowname = "PowerSupply" * name

    # Flows are variables that contribute into Balance equations
    push!(elements, getelement(FLOW_CONCEPT, "BaseFlow", genname))

    # Arrows connect Flows and Balances
    push!(elements, getelement(ARROW_CONCEPT, "BaseArrow", powerarrowname,
        (BALANCE_CONCEPT, balance),
        (FLOW_CONCEPT, genname),
        (CONVERSION_CONCEPT, 1.0), # Factor to convert Flow into the same commodity as Balance (here Power to Power, so 1)
        (DIRECTIONKEY, DIRECTIONIN))) # Positive or negative contribution

    # Cost is the contribution of the variable into the objective function
    costdata = getelement(COST_CONCEPT, "CostTerm", costname,
        (WHICHCONCEPT, FLOW_CONCEPT),
        (WHICHINSTANCE, genname),
        (DIRECTIONKEY, DIRECTIONIN)) # positive or negative contribution
    if cost === nothing # Point to a parameter DataElement with time-series data
        costdata.value[PARAM_CONCEPT] = costname # assume param with same name
    else # If the cost is constant for all scenarios
        cost = float(cost)
        @assert cost >= 0
        costdata.value[PARAM_CONCEPT] = cost
    end
    push!(elements, costdata)

    # Non-negative capacity
    capdata = getelement(CAPACITY_CONCEPT, "PositiveCapacity", capname,
        (WHICHCONCEPT, FLOW_CONCEPT),
        (WHICHINSTANCE, genname),
        (PARAM_CONCEPT, capname), # Point to a parameter DataElement with time-series data
        (BOUNDKEY, BOUNDUPPER)) # Upper or Lower capacity
    push!(elements, capdata)
    if cap != nothing # If the capacity is constant for all scenarios
        cap = float(cap)
        @assert cap >= 0
        # Parameter that converts the capacity in MW to GWh based on the duration of the horizon periods
        push!(elements, getelement(PARAM_CONCEPT, "MWToGWhSeriesParam", capname,
            ("Level", cap),
            ("Profile", 1.0)))
    end
    return
end

# DataElements for a simple reservoir hydropower plant
function addhydro!(elements, name, powerbalance;
    eneq::Union{Real, Nothing}=nothing, # GWh/Mm3 (or kWh/m3)
    releasecap::Union{Real, Nothing}=nothing, # m3/s
    storagecap::Union{Real, Nothing}=nothing) # Mm3
    hydrobalance = name

    releasename = "Release" * name
    spillname = "Spill" * name
    storagename = "Storage" * name

    storagecapname = "Cap" * storagename
    releasecapname = "Cap" * releasename
    inflowname = "Inflow" * hydrobalance

    powerarrowname = "PowerSupply" * releasename
    hydroarrowname = "WaterDemand" * releasename
    spillarrowname = "Spill" * releasename

    # Water balance
    addbalance!(elements, hydrobalance, "Hydro")

    # Inflow is a contribution to the right-hand-side of the water balance
end

```

```

addrhsterm!(elements, inflowname, hydrobalance, DIRECTIONIN)

# Variables for release and spill
push!(elements, getelement(FLOW_CONCEPT, "BaseFlow", releasename))
push!(elements, getelement(FLOW_CONCEPT, "BaseFlow", spillname))

# Release and spill take water out from the hydro balance.
push!(elements, getelement(ARROW_CONCEPT, "BaseArrow", hydroarrowname,
    (BALANCE_CONCEPT, hydrobalance),
    (FLOW_CONCEPT, releasename),
    (CONVERSION_CONCEPT, 1.0),
    (DIRECTIONKEY, DIRECTIONOUT)))

push!(elements, getelement(ARROW_CONCEPT, "BaseArrow", spillarrowname,
    (BALANCE_CONCEPT, hydrobalance),
    (FLOW_CONCEPT, spillname),
    (CONVERSION_CONCEPT, 1.0),
    (DIRECTIONKEY, DIRECTIONOUT)))

# Release also contributes to the power market
powerarrowdata = getelement(ARROW_CONCEPT, "BaseArrow", powerarrowname,
    (BALANCE_CONCEPT, powerbalance),
    (FLOW_CONCEPT, releasename),
    (DIRECTIONKEY, DIRECTIONIN))

if eneq === nothing
    powerarrowdata.value[CONVERSION_CONCEPT] = powerarrowname
else
    eneq = float(neq)
    @assert eneq >= 0
    powerarrowdata.value[CONVERSION_CONCEPT] = eneq
end
push!(elements, powerarrowdata)

# Release capacity
releasecapdata = getelement(CAPACITY_CONCEPT, "PositiveCapacity", releasecapname,
    (WHICHCONCEPT, FLOW_CONCEPT),
    (WHICHINSTANCE, releasename),
    (PARAM_CONCEPT, releasecapname), # Point to a parameter DataElement with time-series data
    (BOUNDKEY, BOUNDUPPER)) # Upper or Lower capacity
push!(elements, releasecapdata)
if releasecap != nothing # If the capacity is constant for all scenarios
    releasecap = float(releasecap)
    @assert releasecap >= 0
    # Parameter that converts the capacity in m3/s to Mm3 based on the duration of the horizon periods
    push!(elements, getelement(PARAM_CONCEPT, "M3SToMM3SeriesParam", releasecapname,
        ("Level", releasecap),
        ("Profile", 1.0)))
end

# Variable for storage
push!(elements, getelement(STORAGE_CONCEPT, "BaseStorage", storagename,
    (BALANCE_CONCEPT, hydrobalance)))

# Storage capacity
storagecapdata = getelement(CAPACITY_CONCEPT, "PositiveCapacity", storagecapname,
    (WHICHCONCEPT, STORAGE_CONCEPT),
    (WHICHINSTANCE, storagename),
    (BOUNDKEY, BOUNDUPPER))
if storagecap === nothing
    storagecapdata.value[PARAM_CONCEPT] = storagecapname
else
    storagecap = float(storagecap)
    @assert storagecap >= 0
    storagecapdata.value[PARAM_CONCEPT] = storagecap
end
push!(elements, storagecapdata)

return
end

# DataElements for transmission between areas
function addpowertrans!(elements, frombalance, tobalance;
    cap::Union{Real, Nothing}=nothing,
    eff::Union{Real, Nothing}=nothing)

flowname = frombalance * "->" * tobalance
capname = "Cap" * flowname
fromarrowname = flowname * "From"
toarrowname = flowname * "To"

# Transmission variable
push!(elements, getelement(FLOW_CONCEPT, "BaseFlow", flowname))

# Variable out from one Balance
fromarrowdata = getelement(ARROW_CONCEPT, "BaseArrow", fromarrowname,
    (BALANCE_CONCEPT, frombalance),
    (FLOW_CONCEPT, flowname),
    (CONVERSION_CONCEPT, 1.0),
    (DIRECTIONKEY, DIRECTIONOUT))
push!(elements, fromarrowdata)

# Variable in to another Balance
toarrowdata = getelement(ARROW_CONCEPT, "BaseArrow", toarrowname,
    (BALANCE_CONCEPT, tobalance),
    (FLOW_CONCEPT, flowname),
    (DIRECTIONKEY, DIRECTIONIN))

```

```

if eff == nothing
    toarrowdata.value[CONVERSION_CONCEPT] = powerarrowname
else
    @assert 0 < eff <= 1
    eff = float(eff)
    toarrowdata.value[CONVERSION_CONCEPT] = eff # this could also be modelled as a Loss
end
push!(elements, toarrowdata)

# Transmission capacity
capdata = getelement(CAPACITY_CONCEPT, "PositiveCapacity", capname,
(WHICHCONCEPT, FLOW_CONCEPT),
(WHICHINSTANCE, flowname),
(PARAM_CONCEPT, capname), # Point to a parameter DataElement with time-series data
(BOUNDKEY, BOUNDUPPER)) # Upper or lower capacity
push!(elements, capdata)
if cap != nothing # If the capacity is constant for all scenarios
    cap = float(cap)
    @assert cap >= 0
    # Parameter that converts the capacity in MW to GWh based on the duration of the horizon periods
    push!(elements, getelement(PARAM_CONCEPT, "MWToGWhSeriesParam", capname,
        ("Level", cap),
        ("Profile", 1.0)))
end
end

printdicts(elements) = JSON.print(elements, 2);

```

## 1.3 - Define dataset for the power market

```

In [94]: # Combine the different parts of the dataset into one list of DataElements
function gettestdataset()
    elements = DataElement[]

    structure = getteststructure()
    elements = vcat(elements, structure)

    params = gettestparams()
    elements = vcat(elements, params)

    constants = gettestconstants()
    elements = vcat(elements, constants)

    levels = gettestlevels()
    elements = vcat(elements, levels)

    profiles = gettestprofiles()
    elements = vcat(elements, profiles)

    return elements
end

# The structure consist of the main model objects and how they are connected together
# We also add some of the parameters (like capacities, conversions and costs) if they are constant for all scenarios
function getteststructure()
    structure = DataElement[]

    addbalance!(structure, "PowerBalance_NO2", "Power")
    addbalance!(structure, "PowerBalance_GER", "Power")

    addpowertrans!(structure, "PowerBalance_NO2", "PowerBalance_GER", cap=1400, eff=0.97)
    addpowertrans!(structure, "PowerBalance_GER", "PowerBalance_NO2", cap=1400, eff=0.97)

    addrhsterm!(structure, "WindNO2", "PowerBalance_NO2", DIRECTIONIN)
    addrhsterm!(structure, "RoRNO2", "PowerBalance_NO2", DIRECTIONIN)
    addrhsterm!(structure, "DemandNO2", "PowerBalance_NO2", DIRECTIONOUT)

    addrhsterm!(structure, "WindGER", "PowerBalance_GER", DIRECTIONIN)
    addrhsterm!(structure, "SolarGER", "PowerBalance_GER", DIRECTIONIN)
    addrhsterm!(structure, "DemandGER", "PowerBalance_GER", DIRECTIONOUT)

    addrhsthermal!(structure, "BioGER", "PowerBalance_GER", cap=5000, cost=50000)
    addrhsthermal!(structure, "NucGER", "PowerBalance_GER", cap=5000, cost=5000)
    addrhsthermal!(structure, "CoalGER", "PowerBalance_GER", cap=15000)
    addrhsthermal!(structure, "GasGER", "PowerBalance_GER", cap=40000)

    addhydro!(structure, "ResNO2", "PowerBalance_NO2", eneq=1.3, storagecap=16000, releasecap=6000)

    return structure
end

# These parameters are built up with TimeVectors (constants or time-series data), which they point to
function gettestparams()
    params = DataElement[]

    # MWToGWhSeriesParam have a profile and a level stored in other DataElements
    # And convert the value from MW to GWh based on the duration of horizon periods
    for name in ["WindNO2", "DemandNO2", "WindGER", "SolarGER", "DemandGER"]
        push!(params, DataElement(PARAM_CONCEPT, "MWToGWhSeriesParam", name,
            Dict("Profile" => "Profile" * name, "Level" => "Level" * name)))
    end

    # M3SToMM3SeriesParam have a profile and a level stored in other DataElements
    # And convert the value from m3/s to Mm3 based on the duration of horizon periods

```

```

for name in ["InflowResNO2", "RoRNO2"]
    push!(params, DataElement(PARAM_CONCEPT, "M3SToMM3SeriesParam", name,
        Dict("Profile" => "Profile" * name, "Level" => "Level" * name)))
end

# FossilMCPParam is calculated from many inputs. Here, most of them are constant, except for the
# gas price Level and profile that is dependant on chosen scenario
for fuel in ["Coal", "Gas"]
    name = "MC" * fuel * "GER"
    push!(params, DataElement(PARAM_CONCEPT, "FossilMCPParam", name,
        Dict("FuelProfile" => "Profile" * fuel, "FuelLevel" => "Level" * fuel,
            "CO2Factor" => "CO2Factor" * fuel, "CO2Profile" => "CO2Profile", "CO2Level" => "CO2Level",
            "Efficiency" => "Efficiency" * fuel, "VOC" => "VOC" * fuel)))
end
params
end

# We define the constant TimeVectors which make up FossilMCPParam
function gettestconstants()
    constants = DataElement[]
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "LevelCoal",
        Dict("Value" => 30000.0))) # €/GWh
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "ProfileCoal",
        Dict("Value" => 1.0)))
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "CO2FactorGas",
        Dict("Value" => 0.18)))
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "CO2FactorCoal",
        Dict("Value" => 0.36)))
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "VOCGas",
        Dict("Value" => 2000.0))) # €/GWh
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "VOCCoal",
        Dict("Value" => 4000.0))) # €/GWh
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "CO2Level",
        Dict("Value" => 50000.0))) # €/GWh
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "CO2Profile",
        Dict("Value" => 1.0)))
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "EfficiencyGas",
        Dict("Value" => 0.5)))
    push!(constants, DataElement(TIMEVECTOR_CONCEPT, "ConstantTimeVector", "EfficiencyCoal",
        Dict("Value" => 0.4)))
    constants
end

# We define TimeVectors that represent the level of different parameters in 2021 and 2025.
function gettestlevels()
    levels = DataElement[]

    push!(levels, DataElement(TIMEINDEX_CONCEPT, "VectorTimeIndex", "DataLevelsTimeIndex",
        Dict("Vector" => [getisoyearstart(2021), getisoyearstart(2025)])))

    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelWindNO2",
        Dict("Vector" => Float64[1500, 1500])) # MW)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelRoRNO2",
        Dict("Vector" => Float64[500, 500])) # MW)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelDemandNO2",
        Dict("Vector" => Float64[5000, 5500])) # MW)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelDemandGER",
        Dict("Vector" => Float64[50000, 55000])) # MW)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelWindGER",
        Dict("Vector" => Float64[30000, 40000])) # MW)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelSolarGER",
        Dict("Vector" => Float64[60000, 80000])) # MW)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelGas",
        Dict("Vector" => Float64[90000, 50000])) # €/GWh)
    push!(levels, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "LevelInflowResNO2",
        Dict("Vector" => Float64[950, 1000])) # m3/s)

    for name in ["LevelWindNO2", "LevelRoRNO2", "LevelDemandNO2", "LevelDemandGER",
        "LevelWindGER", "LevelSolarGER", "LevelGas", "LevelInflowResNO2"]
        push!(levels, DataElement(TIMEVECTOR_CONCEPT, "InfiniteTimeVector", name,
            Dict(TIMEINDEX_CONCEPT => "DataLevelsTimeIndex", TIMEVALUES_CONCEPT => name)))
    end
    levels
end

# We define TimeVectors that represent the profile of different parameters
function gettestprofiles()

    # Profiles from https://www.nve.no/energi/analyser-og-statistikk/verdiasett-for-kraftsystemmodellene/
    path = "testprofiles_1981_2010.csv"
    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)
    colnames = [n for n in names(df) if n != "Timestamp"]
    matrix = Matrix{Float64}(df[:, colnames])
    elements = DataElement[]
    colnames = ["Profile" * name for name in colnames]

    # The time-series data is stored efficiently in two DataElements
    # - RangeTimeIndex is a time series index described by a StepRange
    # - BaseTable stores the data in a matrix, where each column is a different profile,
    # and each row represent the value at a TimeIndex
    push!(elements, DataElement(TIMEINDEX_CONCEPT, "RangeTimeIndex", "ProfilesTimeIndex",
        Dict("Start" => start, "Delta" => Hour(1), "Steps" => numperiods)))
    push!(elements, DataElement(TABLE_CONCEPT, "BaseTable", "ProfilesTable",
        Dict("Matrix" => matrix, "Names" => colnames)))
    for name in colnames

        # ColumnTimeValues points to a row in BaseTable in another DataElement
        push!(elements, DataElement(TIMEVALUES_CONCEPT, "ColumnTimeValues", name,
            Dict(TABLE_CONCEPT => "ProfilesTable", "Name" => name)))
        # RotatingTimeVector points to a index and values pair in other DataElements
        push!(elements, DataElement(TIMEVECTOR_CONCEPT, "RotatingTimeVector", name,
            Dict(TABLE_CONCEPT => "ProfilesTable", "Index" => name)))
    end
end

```

```

    Dict(TIMEVALUES_CONCEPT => name, TIMEINDEX_CONCEPT => "ProfilesTimeIndex")))
end

# We make a simple gas price profile for the variation throughout the year
gasprice = Float64[1.079, 1.037, 0.970, 0.931, 0.915, 0.971, 1.028, 1.070]
datetimes = DateTime[getisoyearstart(1981) + Hour((i-1) * 1113) for i in 1:8]
gasprice = gasprice/mean(gasprice)
push!(elements, DataElement(TIMEINDEX_CONCEPT, "VectorTimeIndex", "IndexProfileGas",
    Dict("Vector" => datetimes)))
push!(elements, DataElement(TIMEVALUES_CONCEPT, "VectorTimeValues", "ValuesProfileGas",
    Dict("Vector" => gasprice)))
push!(elements, DataElement(TIMEVECTOR_CONCEPT, "OneYearTimeVector", "ProfileGas",
    Dict(TIMEVALUES_CONCEPT => "ValuesProfileGas", TIMEINDEX_CONCEPT => "IndexProfileGas")))
return elements
end;

```

## 1.4 - Runsettings

Define some model settings that completes the dataset.

See Demo 2 for the use of a more complex horizon which is built based on the residual load.

```
In [95]: function gettestrunsettings()
    runsettings = DataElement[]

    # 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
    push!(runsettings, getelement(TIMEPERIOD_CONCEPT, "ScenarioTimePeriod", "ScenarioTimePeriod",
        ("Start", getisoyearstart(scenarioyearstart)), ("Stop", getisoyearstart(scenarioyearstop)))))

    # 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)
    global power_horizon = SequentialHorizon(364*3, Day(1))
    global hydro_horizon = SequentialHorizon(52*3, Week(1))
    push!(runsettings, getelement(COMMODITY_CONCEPT, "BaseCommodity", "Power",
        (HORIZON_CONCEPT, power_horizon)))
    push!(runsettings, getelement(COMMODITY_CONCEPT, "BaseCommodity", "Hydro",
        (HORIZON_CONCEPT, hydro_horizon)))

    # 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!(runsettings, getelement(BOUNDARYCONDITION_CONCEPT, "StartEqualStop", "StartEqualStop_StorageResNO2",
        (WHICHINSTANCE, "StorageResNO2"),
        (WHICHCONCEPT, STORAGE_CONCEPT)))

    return runsettings
end

Out[95]: gettestrunsettings (generic function with 1 method)
```

## 1.5 Convert the DataElements into model objects

```
In [100... testdata = gettestdata()
runsettings = gettestrunsettings()

elements = vcat(testdata, runsettings)

@time modelobjects = getmodelobjects(elements);

0.000863 seconds (3.17 k allocations: 133.984 KiB)
```

## 1.6 Look at modelobjects

### Take a look at the dict of modelobjects

```
In [101... display(modelobjects)

Dict{Any, Any} with 15 entries:
Id("Flow", "SlackVarPowe... => BaseFlow(Id("Flow", "SlackVarPowerBalance_N02"),...
Id("Flow", "GenCoalGER") => BaseFlow(Id("Flow", "GenCoalGER"), SequentialHor...
Id("Balance", "PowerBal... => BaseBalance(Id("Balance", "PowerBalance_GER"), B...
Id("Flow", "ReleaseResNO... => BaseFlow(Id("Flow", "ReleaseResNO2"), Sequential...
Id("Flow", "PowerBalance... => BaseFlow(Id("Flow", "PowerBalance_GER->PowerBal...
Id("Balance", "ResNO2") => BaseBalance(Id("Balance", "ResNO2"), BaseCommodi...
Id("Flow", "GenBioGER") => BaseFlow(Id("Flow", "GenBioGER"), SequentialHori...
Id("Flow", "GenNucGER") => BaseFlow(Id("Flow", "GenNucGER"), SequentialHori...
Id("Balance", "PowerBal... => BaseBalance(Id("Balance", "PowerBalance_N02"), B...
Id("Flow", "SpillResNO2") => BaseFlow(Id("Flow", "SpillResNO2"), SequentialHo...
Id("Flow", "SlackVarPowe... => BaseFlow(Id("Flow", "SlackVarPowerBalance_GER"),...
Id("Flow", "GenGasGER") => BaseFlow(Id("Flow", "GenGasGER"), SequentialHori...
Id("BoundaryCondition", ... => StartEqualStop(Id("BoundaryCondition", "StartEqu...
Id("Storage", "StorageRe... => BaseStorage(Id("Storage", "StorageResNO2"), Base...
Id("Flow", "PowerBalance... => BaseFlow(Id("Flow", "PowerBalance_N02->PowerBal..."
```

### Take a look at the hydropower release variables

```
In [102...]
display("We display the fields in the struct")
display(fieldnames(typeof(modelobjects[Id("Flow", "ReleaseResNO2")])))
display("The Id shows that the struct is of type Flow")
display(getid(modelobjects[Id("Flow", "ReleaseResNO2")]))
display("The Horizon consist of 1092 periods with timedelta of 1 day (total 3 years)")
display(gethorizon(modelobjects[Id("Flow", "ReleaseResNO2")]))
display("The lower bound is LowerZeroCapacity so that the variable is non-negative")
display(getlb(modelobjects[Id("Flow", "ReleaseResNO2")]))
display("The upper bound is 6000 m3/s, which is stored in a M3SToMM3SeriesParam. M3SToMM3SeriesParam converts the upper bound into Mm")
display(getub(modelobjects[Id("Flow", "ReleaseResNO2")]))
display("There are two arrows. The first says that the release takes water out from the water balance and the second says that it cor")
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 depen"
ding 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 contribute"
s into the power balance"
Id("Arrow", "WaterDemandReleaseResNO2")
Id("Arrow", "PowerSupplyReleaseResNO2")
```

## Take a look at the wind parameter for GER

```
In [103...]
# It is stored in one of the rhsterms of PowerBalance_NO2
rhsterm = getrhsterms(modelobjects[Id("Balance", "PowerBalance_GER")])[1]

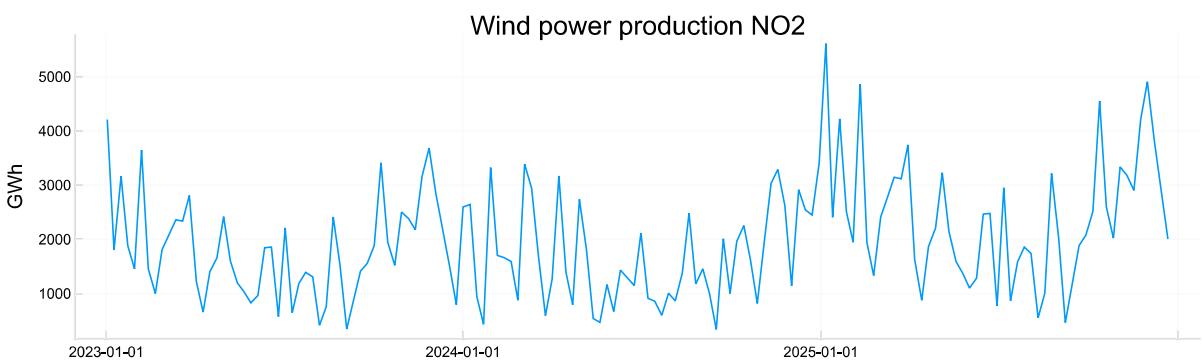
# We start with the installed capacity in 2023 and the weather scenario 1981
start = TwoTime(getisoyearstart(2023), getisoyearstart(1981))

# 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=(900,300), ylabel = "GWh", title = "Wind power production NO2"))
```



- 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 [104]: mymodel = JuMP.Model(HiGHS.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[104]: 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: HiGHS  
Names registered in the model: BalancePowerBalance\_GER, BalancePowerBalance\_N02, BalanceResN02, BoundaryConditionEqStartEqualStop\_St  
orangeResN02, FlowGenBioGER, FlowGenCoalGER, FlowGenGasGER, FlowGenNucGER, FlowPowerBalance\_GER->PowerBalance\_N02, FlowPowerBalance\_N  
02->PowerBalance\_GER, FlowReleaseResN02, FlowSlackVarPowerBalance\_GER, FlowSlackVarPowerBalance\_N02, FlowSpillResN02, StorageStartSt  
orangeResN02, StorageStorageResN02

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

```
In [105]: display(prob.model[:BalanceResN02][2])
```

BalanceResN02[2] :  
 $-FlowReleaseResNO2_8 - FlowReleaseResNO2_9 - FlowReleaseResNO2_{10} - FlowReleaseResNO2_{11} - FlowReleaseResNO2_{12} - Flow$   
 $+ StorageStorageResNO2_1 = 0.0$

## 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 [106]: t = TwoTime(getisoyearstart(2021), getisoyearstart(1981))
update!(prob, t)
```

## 2.3 - Solve the problem and display the solution

```
In [107]: @time solve!(prob)
getobjectivevalue(prob)
0.097816 seconds (240.47 k allocations: 10.907 MiB)
9.616463034033783e10
```

Out[107]:

Now the hydropower balance also includes the inflow, which is 156 Mm<sup>3</sup> in the second week of the problem.

```
In [108]: display(prob.model[:BalanceResN02][2])
```

BalanceResN02[2] :  
 $-FlowReleaseResNO2_8 - FlowReleaseResNO2_9 - FlowReleaseResNO2_{10} - FlowReleaseResNO2_{11} - FlowReleaseResNO2_{12} - Flow$   
 $+ StorageStorageResNO2_1 = -156.153096$

## 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 [109]: t = TwoTime(getisoyearstart(2024), getisoyearstart(1982))
update!(prob, t)
solve!(prob)
display(getobjectivevalue(prob))
```

```

display(prob.model[:BalancePowerBalance_GER][2])
8.19444102290675e10
BalancePowerBalance_GER[2] :
FlowGenCoalGER2 - FlowPowerBalance_GER -> PowerBalance_NO2 + FlowGenBioGER2 + FlowGenNucGER2 - FlowSlackVa

```

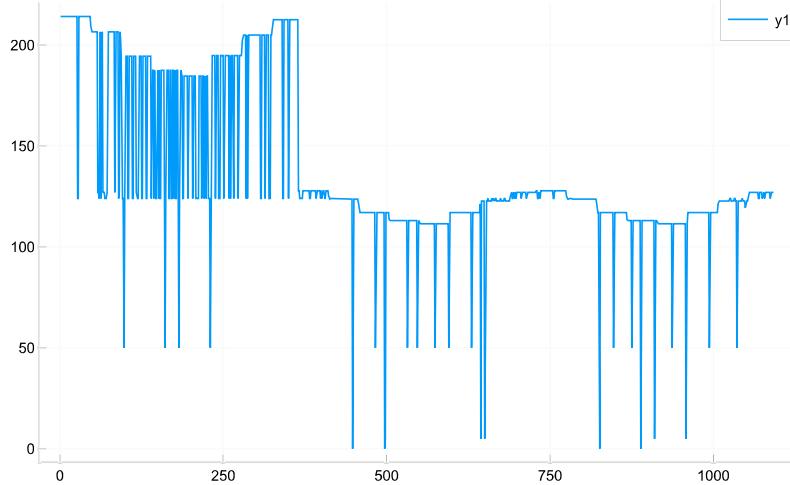
## 2.5 - Look at prices

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

Out[110]:

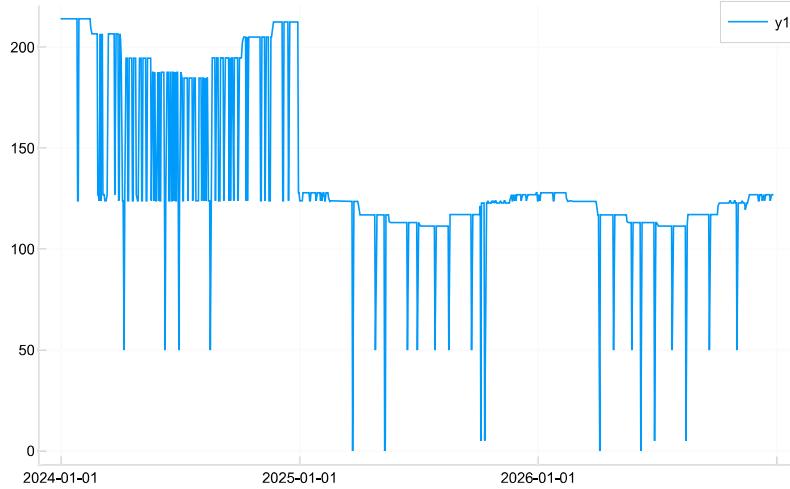


### Or we could use the result functions built into JuMP\_Prob

- getdual() to get prices
- getvarvalue()
- getrhs()

```
In [111]: numperiods = getnumperiods(power_horizon)
x = [getdatetime(t) + getstartduration(power_horizon, s) for s in 1:numperiods]
y = [getdual(prob, Id("Balance", "PowerBalance_GER"), s) for s in 1:numperiods]/1000
plot(x,y)
```

Out[111]:



## 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 [112]: # Make List of scenarios
function getscenarios(dt; years)
    [TwoTime(getisoyearstart(dt), getisoyearstart(yr)) for yr in years]
end

# Run scenarios and plot results from the whole problem
function runscenarios(scenarios, modelobjects)
    runscenarios(scenarios, modelobjects, values(modelobjects))
end

# Run scenarios and plot results from chosen model objects (e.g. only one price area)
function runscenarios(scenarios, modelobjects, resultobjects)
    numperiods_powerhorizon = getnumperiods(power_horizon)
    numperiods_hydrohorizon = getnumperiods(hydro_horizon)

    # We use the datatime for plotting results
    dt = getdatatime(scenarios[1])

    # Time vector for power_horizon
    x1 = [dt + getstartduration(power_horizon, t) for t in 1:numperiods_powerhorizon]

    # Time vector for hydro_horizon
    x2 = [dt + getstartduration(hydro_horizon, t) for t in 1:numperiods_hydrohorizon]

    # Problem can be a HiGHS_Prob or a JuMP_Prob
    # prob = HiGHS_Prob(collect(values(modelobjects)))
    model = Model(HiGHS.Optimizer)

    set_silent(model)
    prob = JuMP_Prob(collect(values(modelobjects)), model)

    # Order result objects into lists
    powerbalances = []
    rhsterms = []
    plants = []
    plantbalances = []
    planarrows = Dict()
    demands = []
    demandbalances = []
    demandarrows = Dict()
    hydrostorages = []

    for obj in resultobjects

        # Powerbalances
        if obj isa BaseBalance
            if getinstancename(getid(getcommodity(obj))) == "Power"
                push!(powerbalances, getid(obj))
                for rhsterm in getrhsterms(obj)
                    push!(rhsterms, getid(rhsterm))
                end
            end
        end

        # Hydrostorages
        if obj isa BaseStorage
            if getinstancename(getid(getcommodity(getbalance(obj)))) == "Hydro"
                push!(hydrostorages, getid(obj))
            end
        end

        # Supply and demands
        if obj isa BaseFlow
            # The type of supply or demand can be found based on the arrows
            arrows = getarrows(obj)

            # Simple supplies and demands
            powerarrowbool = [getid(getcommodity(getbalance(arrow))) == Id("Commodity", "Power") for arrow in arrows]
            powerarrows = arrows[powerarrowbool]
            if sum(powerarrowbool) == 1
                if isingoing(powerarrows[1])
                    push!(plants, getid(obj))
                    push!(plantbalances, getid(getbalance(powerarrows[1])))
                elseif !isingoing(powerarrows[1])
                    push!(demands, getid(obj))
                    push!(demandbalances, getid(getbalance(powerarrows[1])))
                end
            end
        end

        # Transmissions
        if sum(powerarrowbool) == 2
            for arrow in arrows
                balance = getbalance(arrow)
                if getid(getcommodity(balance)) == Id("Commodity", "Power")
                    if isingoing(arrow) && (balance in resultobjects)
                        push!(plants, getid(obj))
                        push!(plantbalances, getid(balance))
                    elseif !isingoing(arrow) && (balance in resultobjects)
                        push!(demands, getid(obj))
                        push!(demandbalances, getid(balance))
                    end
                end
            end
        end
    end
end
```

```

        end
    end
end
end
end
end

# Matrices to store results per time period, scenario and object
prices = zeros(numperiods_powerhorizon, length(scenarios), length(powerbalances))
rhstermvalues = zeros(numperiods_powerhorizon, length(scenarios), length(rhsterms))
production = zeros(numperiods_powerhorizon, length(scenarios), length(plants))
consumption = zeros(numperiods_powerhorizon, length(scenarios), length(demands))
hydrolevels = zeros(numperiods_hydrohorizon, length(scenarios), length(hydrostorages))

# Update and solve scenarios, and collect results
for (s, t) in enumerate(scenarios)
    update!(prob, t)

    solve!(prob)

    println("Objective value in scenario $(s): ", getobjectivevalue(prob))
    write_to_file(prob.model, "model.mps") # can write JuMP model to file

    # Collect results for power production/demand/transmission in GW
    for j in 1:numperiods_powerhorizon

        # Timefactor transform results from GWh to GW/h regardless of horizon period durations
        timefactor = getduration(gettimedelta(power_horizon, j))/Millisecond(3600000)

        # For powerbalances collect prices and rhsterms (Like inelastic demand, wind, solar and RoR)
        for i in 1:length(powerbalances)
            prices[j, s, i] = getconduial(prob, powerbalances[i], j)/1000 # from €/GWh to €/MWh
            for k in 1:length(rhsterms)
                if hasrhterm(prob, powerbalances[i], rhsterms[k], j)
                    rhstermvalues[j, s, k] = getrhterm(prob, powerbalances[i], rhsterms[k], j)/timefactor
                end
            end
        end

        # Collect production of all plants
        for i in 1:length(plants) # TODO: Balance and variable can have different horizons
            production[j, s, i] = getvarvalue(prob, plants[i], j)*abs(getconcoeff(prob, plantbalances[i], plants[i], j, j))/timefactor
        end

        # Collect demand of all demands
        for i in 1:length(demands) # TODO: Balance and variable can have different horizons
            consumption[j, s, i] = getvarvalue(prob, demands[i], j)*abs(getconcoeff(prob, demandbalances[i], demands[i], j, j))/timefactor
        end
    end

    # Collect hydro storage levels
    for j in 1:numperiods_hydrohorizon
        for i in 1:length(hydrostorages)
            hydrolevels[j, s, i] = getvarvalue(prob, hydrostorages[i], j)/1000 # Gm3 TODO: convert to TWh with global energy equ
        end
    end
end

# Only keep rhsterms that have at least one value (TODO: Do the same for supply and demands)
rhstermtotals = dropdims(sum(rhstermvalues,dims=(1,2)),dims=(1,2))
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]*-1
rhstermdemandvalues = rhstermvalues[:, :, rhstermdemandidx]

rhstermsupplynames = [getinstancename(rhsterm) for rhsterm in rhsterms[rhstermsupplyidx]]
rhstermdemandnames = [getinstancename(rhsterm) for rhsterm in rhsterms[rhstermdemandidx]]

supplynames = [[getinstancename(plant) for plant in plants];rhstermsupplynames]
supplyvalues = reshape([vcat(production...);vcat(rhstermsupplyvalues...)],(numperiods_powerhorizon,length(scenarios),length(suppl

demandnames = [[getinstancename(demand) for demand in demands];rhstermdemandnames]
demandvalues = reshape([vcat(consumption...);vcat(rhstermdemandvalues...)],(numperiods_powerhorizon,length(scenarios),length(dem

# Prepare for plotting results
hydronames = [getinstancename(hydro) for hydro in hydrostorages]
powerbalancenames = [split(getinstancename(powerbalance), "PowerBalance_")[2] for powerbalance in powerbalances]

# Plot results for each scenario
for (s, t) in enumerate(scenarios)
    scenyear = string(getisoyear(getscenariotime(t)))
    datayear = string(getisoyear(getdatatime(t)))

    # Plot prices
    display(plot(x1, sum(prices[:,s,:]), dims=3), labels=reshape(powerbalancenames,1,length(powerbalancenames)), size=(800,500), t

```

```

# Plot supplies and demands
supplychart = areaplot(x1, sum(supplyvalues[:,s,:],dims=3),labels=reshape(supplynames,1,length(supplynames)),title="Supply",
demandchart = areaplot(x1, sum(demandvalues[:,s,:],dims=3),labels=reshape(demandnames,1,length(demandnames)),title="Demand",
display(plot([supplychart,demandchart]...,layout=(1,2),size=(1600,500)))

# Plot storages (only TWh because of our input data)
display(areaplot(x2, sum(hydrolevels[:,s,:],dims=3),labels=reshape(hydronames,1,length(hydronames)),size=(800,500),title="Reservoirs")

# Plot List of yearly mean production and demand for each supply/demand
meandemand = dropdims(mean(demandvalues[:,s,:],dims=1),dims=1)
meanproduction = dropdims(mean(supplyvalues[:,s,:],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)

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

end
end;

```

## 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

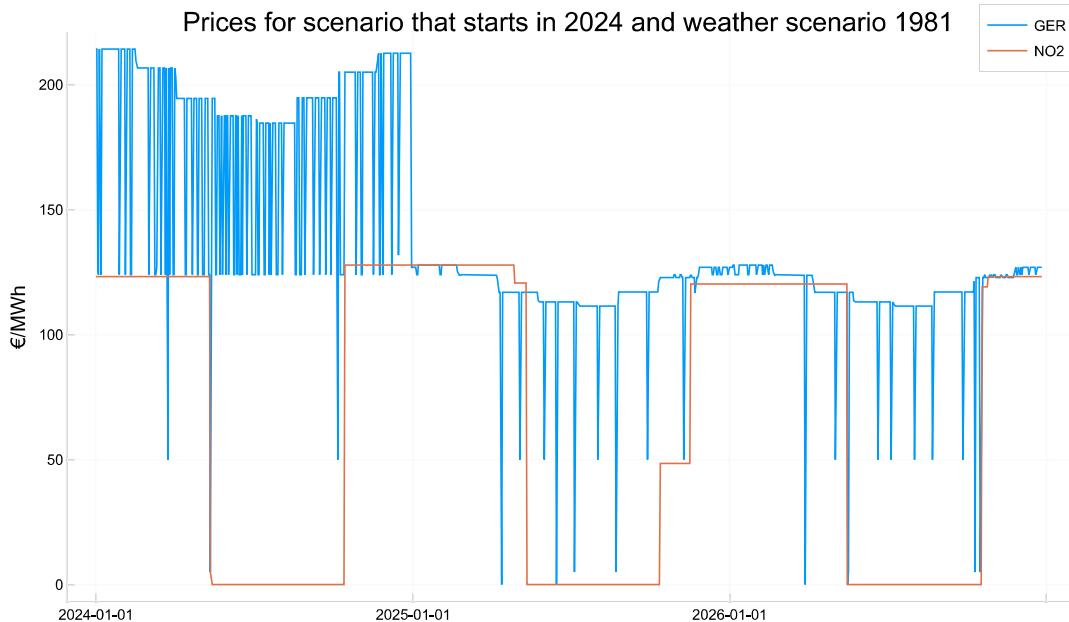
- The supply equals the demand at all times. When the RoR production is very high during the spring flood, the slack variable in the power balance is used as a demand.
- The different scenarios gives prices decided by the marginal cost of the most expensive production technology at any time.
- The hydro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

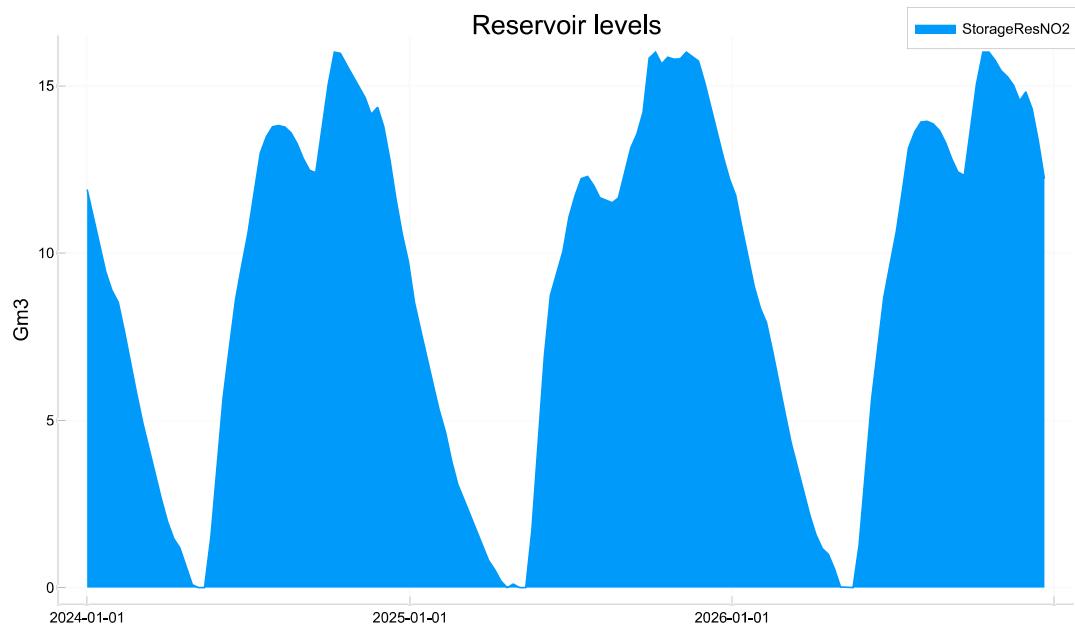
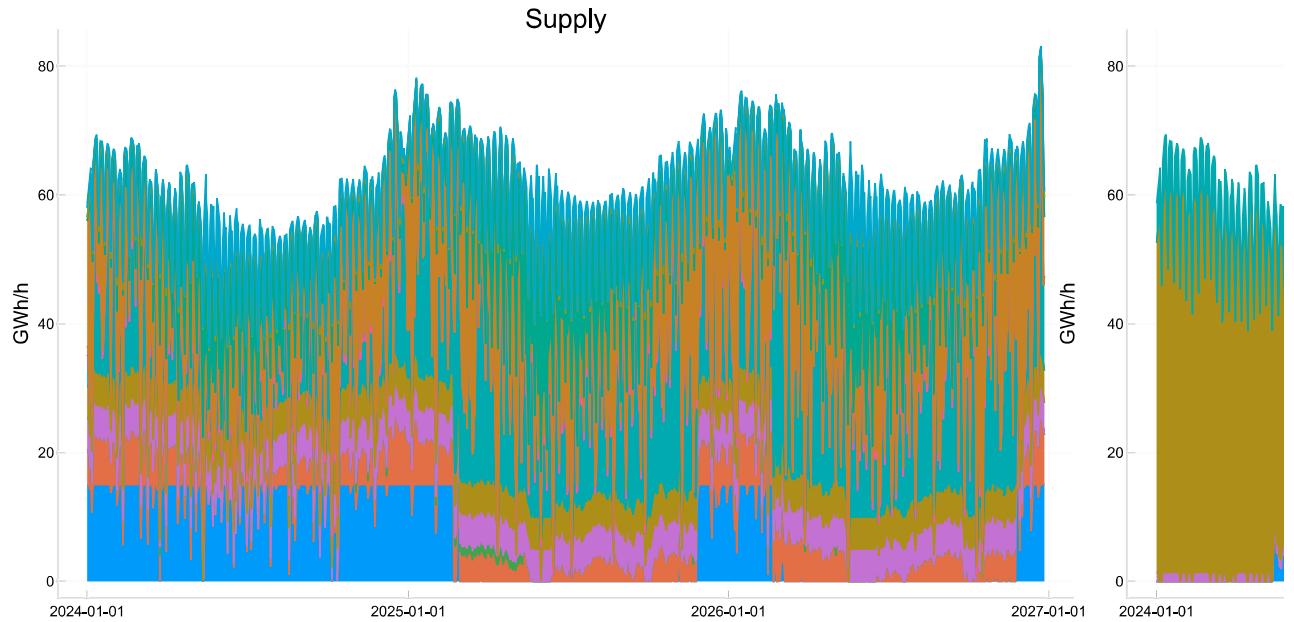
```
In [113...]
scenarios = getscenarios(2024; years=1981:1983)

runscenarios(scenarios, modelobjects)

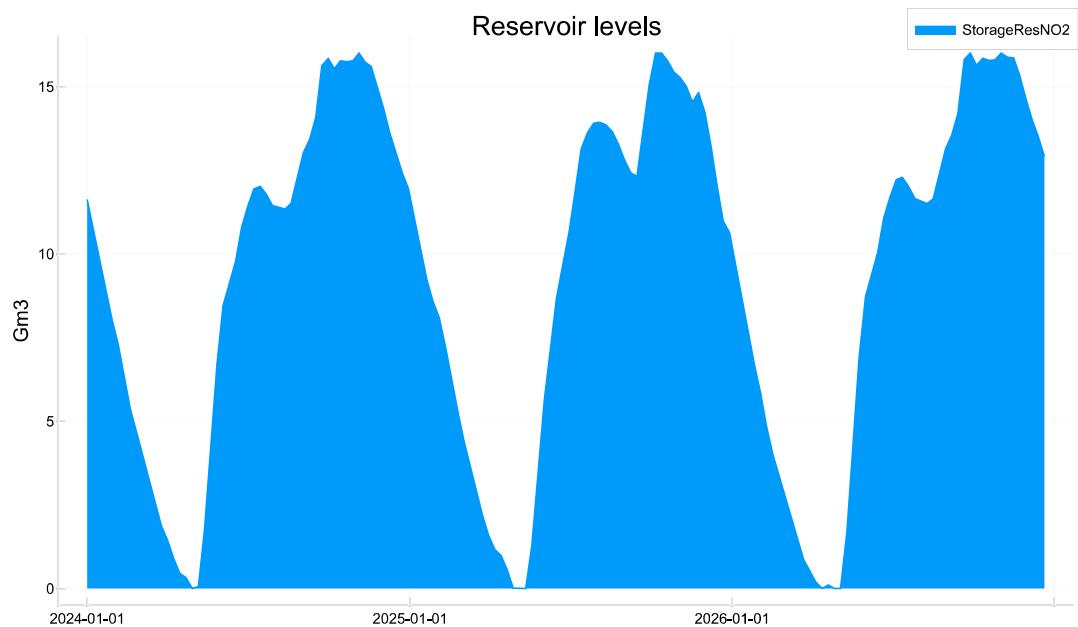
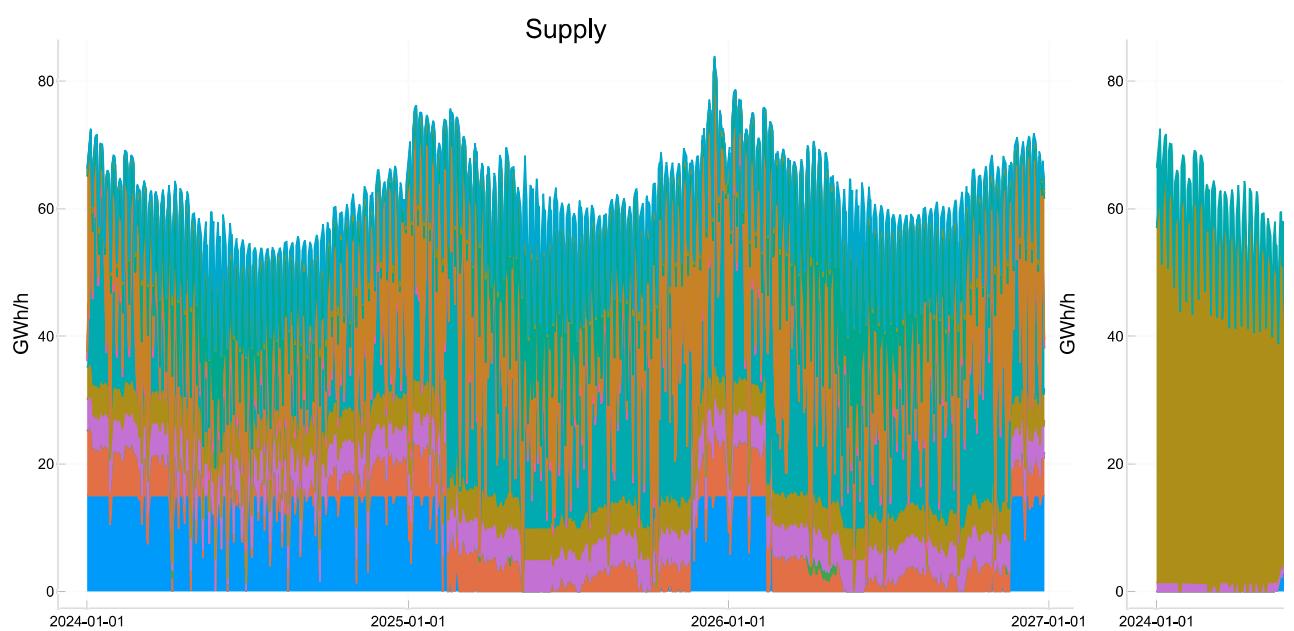
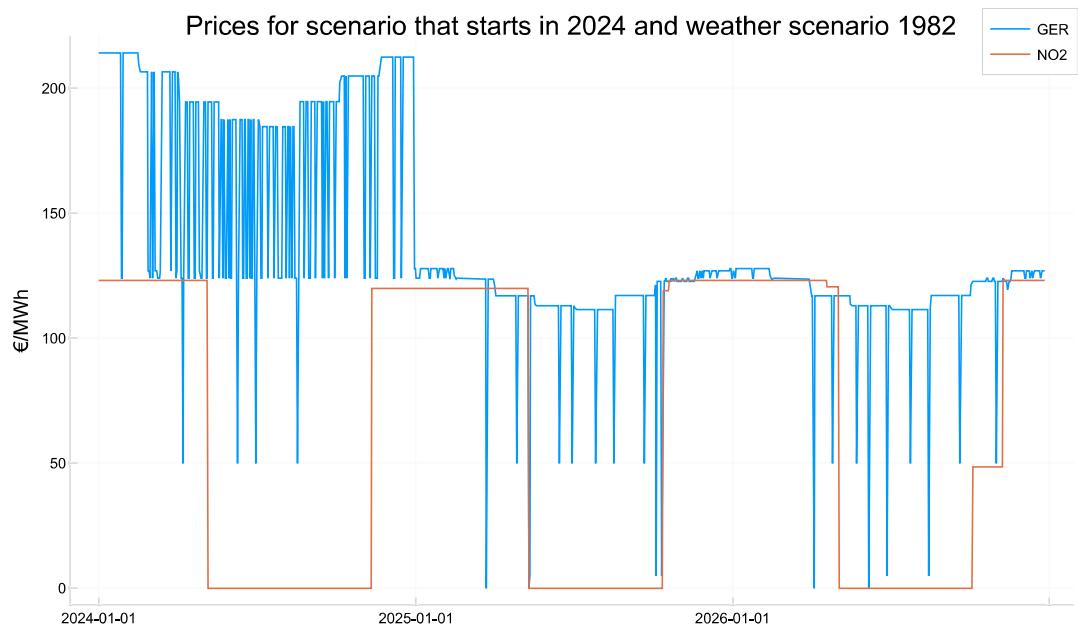
# alternative for only seeing results from one price area
# resultobjects = getpowerobjects(modelobjects,["NO2"])
# runscenarios(scenarios, modelobjects, resultobjects)

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

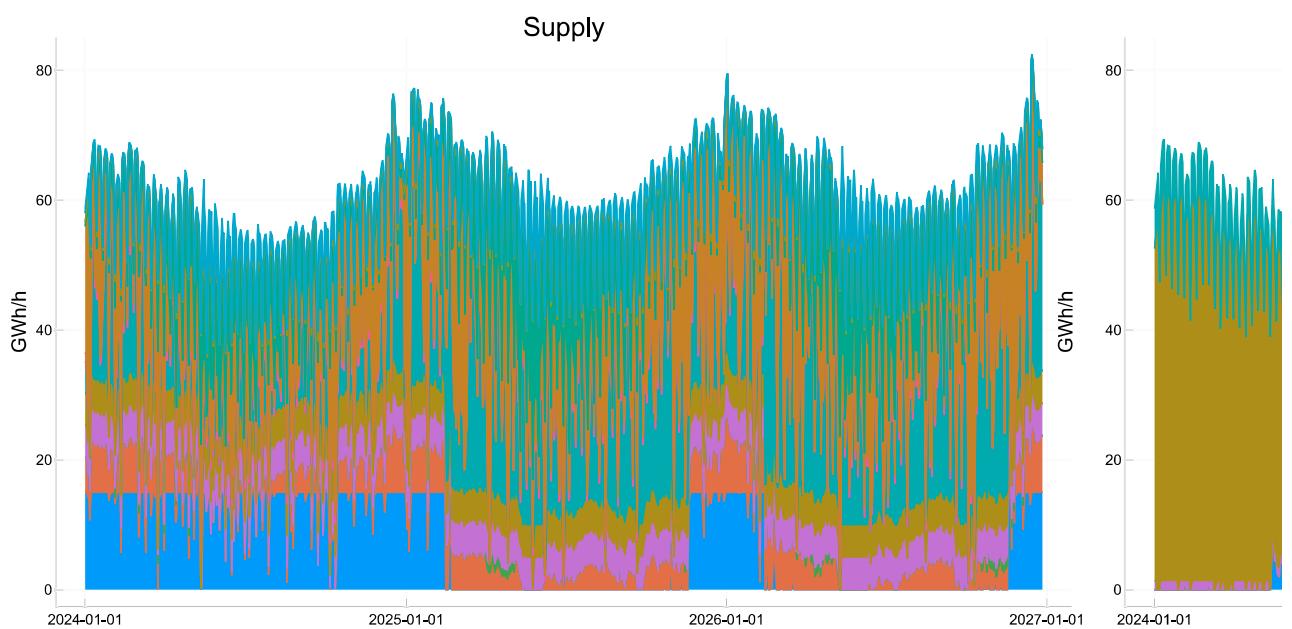
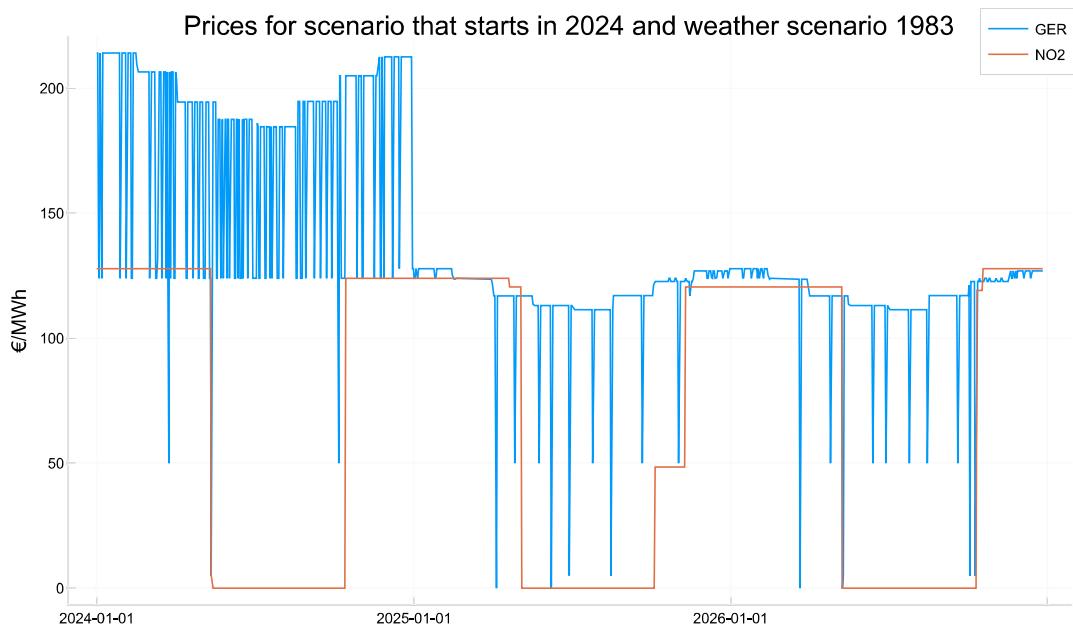


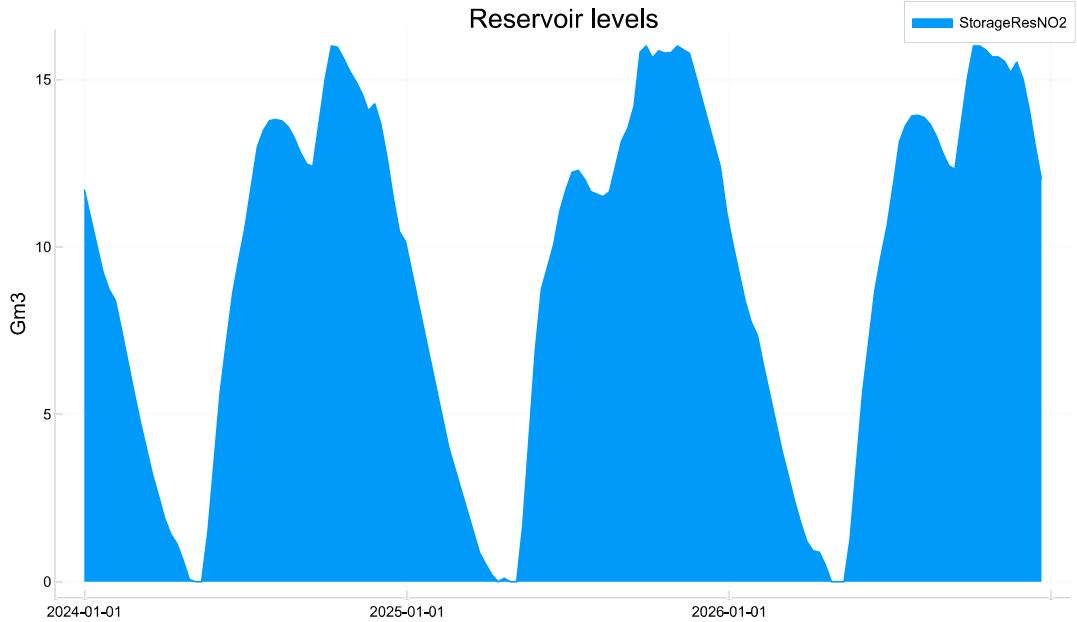


Row	Supplyname String?	Yearly_supply_TWh Float64?	Demandname String?	Yearly_demand_TWh Float64?	
1	GenGasGER	131.058	DemandGER	468.054	
2	WindGER	111.944	DemandNO2	48.4256	
3	SolarGER	70.6705	PowerBalance_NO2->PowerBalance_G...	8.62053	
4	GenCoalGER	60.4118	SlackVarPowerBalance_NO2	1.80325	
5	GenNucGER	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	
Row	Yearly_supply_TWh_sum_skipmissing Float64	Yearly_demand_TWh_sum_skipmissing Float64			1x2 DataFrame
1	527.929	527.929			



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_N02->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_N02->PowerBalance_G...	8.82832	missing	missing
10	WindNO2	5.13809	missing	missing
11	PowerBalance_GER->PowerBalance_N...	0.399974	missing	missing
Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing		1x2 DataFrame
	Float64	Float64		
1		525.968	525.968	





Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
Row	Supplyname	Yearly_supply_TWh	Demandname	Yearly_demand_TWh
1	GenGasGER	131.754	DemandGER	468.242
2	WindGER	111.142	DemandNO2	48.4851
3	SolarGER	70.6877	PowerBalance_N02->PowerBalance_G...	8.56625
4	GenCoalGER	60.7569	SlackVarPowerBalance_N02	1.79717
5	GenNucGER	43.5204	PowerBalance_GER->PowerBalance_N...	0.994443
6	GenBioGER	43.1119	SlackVarPowerBalance_GER	0.0463198
7	ReleaseResN02	36.9577	missing	missing
8	RoRNO2	15.9212	missing	missing
9	PowerBalance_N02->PowerBalance_G...	8.30926	missing	missing
10	WindNO2	5.005	missing	missing
11	PowerBalance_GER->PowerBalance_N...	0.96461	missing	missing
Row	Yearly_supply_TWh_sum_skipmissing	Yearly_demand_TWh_sum_skipmissing		
1	528.131	528.131		

1x2 DataFrame

In [ ]: