Tutorial VII - Storage Modeling

Energy System Optimization with Julia

Dr. Tobias Cors

1. Modeling the Unit Commitment Problem with Storage

Implement the Unit Commitment problem with storage from the lecture in Julia. Before we start, let's load the necessary packages and data.

```
using JuMP, HiGHS
using CSV
using DataFrames
using CairoMakie
using Dates

# Set up CairoMakie
set_theme!(theme_light())
```

1.1 Load and Process Data

First, we load the data from the CSV files and process them into dictionaries for easy access.

```
# Get the directory of the current file
file_directory = "$(@__DIR__)/data"

# Load data
dfGenerators = CSV.read("$file_directory/generator.csv", DataFrame)
dfStorages = CSV.read("$file_directory/storage.csv", DataFrame)
dfWindTurbines = CSV.read("$file_directory/windTurbine.csv", DataFrame)
dfScenarios = CSV.read("$file_directory/scenario.csv", DataFrame)

# Process generator data
dictGenerators = Dict(
    row.name => (
        min_power = row.min_power,
        max_power = row.max_power,
        variable_cost = row.variable_cost,
        fix_cost = row.fix_cost,
        min_up_time = row.min_up_time,
        min_down_time = row.min_down_time,
        ramp_up = row.ramp_up,
        ramp_down = row.ramp_down,
        startup_cost = row.startup_cost,
        efficiency = row.efficiency
    ) for row in eachrow(dfGenerators)
```

```
)

# Process storage data
dictStorages = Dict(
    row.name => (
        min_power = row.min_power,
        max_power = row.max_power,
        min_energy = row.min_energy,
        max_energy = row.max_energy,
        charge_efficiency = row.charge_efficiency,
        discharge_efficiency = row.discharge_efficiency,
        self_discharge_rate = row.self_discharge_rate,
        ramp_up = row.ramp_up,
        ramp_down = row.ramp_down
    ) for row in eachrow(dfStorages)
)

# Process wind turbine data
dictWindTurbines = Dict(
    row.name => (
        variable_cost = row.variable_cost,
    ) for row in eachrow(dfWindTurbines)
)

# Process scenario data
date_format = dateformat"yyyy-mm-dd HH:MM:SS"
dictScenarios = Dict()
for scenario in unique(dfScenarios.scenario)
    scenario_data = dfScenarios[dfScenarios.scenario .== scenario, :]
    dictScenarios[scenario] = (
        datetime = DateTime.(scenario_data.datetime, date_format),
        demand_forecast = scenario_data.demand_forecast,
        wind_forecast = scenario_data.wind_forecast
    )
end
```

1.2 Implement the Unit Commitment Model with Storage

Now, let's implement the Unit Commitment model with storage. We'll create a function that takes the data dictionaries as input and returns the model.

TASK: Fill out the code below where you see `## YOUR CODE HERE` to implement the storage in the model.

```
function solve_unit_commitment(dictGenerators, dictStorages, dictWindTurbines, scenario)
    # Create model
    model = Model(HiGHS.Optimizer)
    set_silent(model)

    # Define the time periods and sets
    T = 1:length(scenario.datetime)  # Time periods (hours)
    G = keys(dictGenerators)  # Set of thermal generators
    S = keys(dictStorages)    # Set of storage units
    W = keys(dictWindTurbines) # Set of wind turbines
```

```
# Decision Variables
@variable(model, p[g in G, t in T] >= 0)  # Power output of generator g at time t
@variable(model, u[g in G, t in T], Bin)  # Binary variable for generator status
 ↪  (1=on, 0=off)
@variable(model, v[g in G, t in T], Bin)  # Binary variable for startup (1=startup,
 ↪  0=no startup)
@variable(model, p_w[w in W, t in T] >= 0)  # Power output of wind at time t
@variable(model, p_fictive[t in T] >= 0)  # Fictive power at time t -> used to model
 ↪  a fictive production in power balance constraint and penalize it with a very high
 ↪  cost in the objective function in case the scenario is not feasible, i.e. not
 ↪  enough generation is available to cover the demand

# Storage variables
@variable(model, p_ch[s in S, t in T] >= 0)     # Charging power of storage s at time
 ↪  t
@variable(model, p_dis[s in S, t in T] >= 0)    # Discharging power of storage s at
 ↪  time t
@variable(model, e[s in S, t in T] >= 0)        # Energy level of storage s at time t
@variable(model, u_ch[s in S, t in T], Bin)     # Binary variable for charging status
 ↪  (1=charging, 0=not charging)
@variable(model, u_dis[s in S, t in T], Bin)    # Binary variable for discharging
 ↪  status (1=discharging, 0=not discharging)

# Objective Function
@objective(model, Min, sum(
    dictGenerators[g].variable_cost * p[g,t] +   # Variable cost of production
    dictGenerators[g].fix_cost * u[g,t] +        # Fixed cost of running
    dictGenerators[g].startup_cost * v[g,t]      # Startup cost of starting the
↪  generator
    for g in G, t in T
) + sum(
    dictWindTurbines[w].variable_cost * p_w[w,t]    # Variable cost of wind
↪  production
    for w in W, t in T
) + sum(
    1000 * p_fictive[t]    # Cost of fictive production
    for t in T
))

# Constraints
# Power balance constraint (including storage): Total generation must equal demand
## YOUR CODE HERE


# Generator limits: Power output must be within min/max when running
@constraint(model, [g in G, t in T],
    p[g,t] <= dictGenerators[g].max_power * u[g,t] # Max power when running
)
@constraint(model, [g in G, t in T],
    p[g,t] >= dictGenerators[g].min_power * u[g,t] # Min power when running
)
```

```
# Wind limits: Wind power cannot exceed forecast
@constraint(model, [w in W, t in T],
    p_w[w,t] <= scenario.wind_forecast[t]
)

# Minimum up time: Generator must stay on for minimum duration after startup
@constraint(model, min_up[g in G, t in T],
    sum(u[g, ] for   in max(1, t-dictGenerators[g].min_up_time+1):t) >=
    dictGenerators[g].min_up_time * v[g,t]
)

# Minimum down time: Generator must stay off for minimum duration after shutdown
@constraint(model, min_down[g in G, t in T],
    sum(1 - u[g, ] for   in max(1, t-dictGenerators[g].min_down_time+1):t) >=
    dictGenerators[g].min_down_time * (1 - u[g,t])
)

# Ramp rate limits: Power change between consecutive timesteps/hours is limited
@constraint(model, [g in G, t in 2:length(T)],
    p[g,t] - p[g,t-1] <= dictGenerators[g].ramp_up      # Max ramp up
)
@constraint(model, [g in G, t in 2:length(T)],
    p[g,t-1] - p[g,t] <= dictGenerators[g].ramp_down    # Max ramp down
)

# Startup variable definition: v_g[g,t] = 1 if generator g is started at time t
@constraint(model, [g in G, t in 2:length(T)],
    v[g,t] >= u[g,t] - u[g,t-1]     # v_g = 1 if u_g changes from 0 (t-1) to 1 (t)
)

# Storage constraints
# Energy balance (Tip: start at t=2)
## YOUR CODE HERE


# Energy balance at t=1: Initial energy level (assume empty at start) (Tip: define
 ↪  the constraint for e[s,t] at t=1)
## YOUR CODE HERE


# Energy limits: Energy level must be within min/max
## YOUR CODE HERE


# Power limits and mutual exclusion: Storage power cannot exceed max power when
 ↪  charging/discharging and charging and discharging cannot happen at the same time
## YOUR CODE HERE


# Storage ramp rate limits: Power change between consecutive timesteps/hours is
 ↪  limited (Tip: define the constraints starting at t=2)
```

```
    # Solve the model
    optimize!(model)

    # Assert that the solution is feasible
    if termination_status(model) != MOI.OPTIMAL
        ts = termination_status(model)
        @info "Optimization finished. The model was not solved correctly. Termination
         ↪  Status: $ts"
        # Helpful resource: https://jump.dev/JuMP.jl/stable/manual/solutions/#Conflicts
    end

    # Return results
    return (
        p_g = value.(p),      # Generator power output
        p_w = value.(p_w),    # Wind power output
        u_g = value.(u),      # Generator status
        v_g = value.(v),      # Startup events
        p_ch = value.(p_ch),  # Storage charging power
        p_dis = value.(p_dis),# Storage discharging power
        e = value.(e),        # Storage energy level
        total_cost = objective_value(model)
    )
end
```

1.3 Solve and Analyze Results

Now, let's solve the model and analyze the results with simple plotting.

```
# Create a dataframe to store results
results_df = DataFrame(
    scenario = String[],        # Scenario identifier
    datetime = DateTime[],      # Timestamp
    total_cost = Float64[],     # Total system cost
    wind_curtailment = Float64[], # Curtailed wind power
    thermal_generation = Float64[], # Total thermal generation
    wind_generation = Float64[],    # Total wind generation
    storage_charge = Float64[],     # Storage charging power
    storage_discharge = Float64[],  # Storage discharging power
    storage_energy = Float64[]      # Storage energy level
)

# Loop over scenarios
for (scenario_name, scenario_data) in dictScenarios
    solution = solve_unit_commitment(dictGenerators, dictStorages, dictWindTurbines,
 ↪  scenario_data)

    # Store results for each time period
    for t in 1:length(scenario_data.datetime)
        push!(results_df, (
            scenario_name,
            scenario_data.datetime[t],
            solution.total_cost,
```

```
            sum(scenario_data.wind_forecast[t] - solution.p_w[w,t] for w in
             ↪ keys(dictWindTurbines)),
            sum(solution.p_g[g,t] for g in keys(dictGenerators)),
            sum(solution.p_w[w,t] for w in keys(dictWindTurbines)),
            sum(solution.p_ch[s,t] for s in keys(dictStorages)),
            sum(solution.p_dis[s,t] for s in keys(dictStorages)),
            sum(solution.e[s,t] for s in keys(dictStorages))
        ))
    end
end

# Plot generation over time for each scenario
for (scenario_name, scenario_data) in dictScenarios
    # Create figure with subplots
    fig = Figure(size=(1000, 800))

    # Format datetime to show only hours
    hours = hour.(results_df.datetime)

    # Generation profile
    ax1 = Axis(fig[1, 1], xlabel="Hour of Day", ylabel="Power [MW]")
    lines!(ax1, hours, results_df.thermal_generation, label="Thermal Generation")
    lines!(ax1, hours, results_df.wind_generation, label="Wind Generation")
    lines!(ax1, hours, results_df.wind_curtailment, label="Wind Curtailment")
    lines!(ax1, hours, scenario_data.wind_forecast, label="Wind Forecast")
    lines!(ax1, hours, scenario_data.demand_forecast, label="Demand")
    axislegend(ax1)
    ax1.title = "Generation Profile for Scenario $scenario_name"

    # Storage energy level
    ax2 = Axis(fig[1, 2], xlabel="Hour of Day", ylabel="Energy [MWh]")
    lines!(ax2, hours, results_df.storage_energy, label="Energy Level")
    axislegend(ax2)
    ax2.title = "Storage Energy Level for Scenario $scenario_name"

    # Storage power
    ax3 = Axis(fig[2, 1], xlabel="Hour of Day", ylabel="Power [MW]")
    lines!(ax3, hours, results_df.storage_charge, label="Charging")
    lines!(ax3, hours, results_df.storage_discharge, label="Discharging")
    axislegend(ax3)
    ax3.title = "Storage Power for Scenario $scenario_name"

    # Display the figure
    display(fig)
end
```

1.4 Verify Results

```
# Test your answer
# Check objective value / total cost is in the correct range
@assert isapprox(results_df.total_cost[1], 1.156e6, atol=1e4) "The total cost for
 ↪ scenario S1 should be 1.156e6 but is $(results_df.total_cost[1])"
```

```
println("Excellent work! You've successfully implemented the storage model and solved the
 ↪  optimization problem.")
```

Solutions

You will likely find solutions to most exercises online. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them in next week's tutorial. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.