The Costs of Counterparty Risk in Long Term Contracts Code Guide - Section 1

Michael Duarte Gonçalves

July 4, 2025

Overview

This file provides a detailed overview of the code structure for the numerical methods exercise. It is intended to guide future contributors through the logic, organization, and rationale of each section and function.

Contents

1 Dataset Creation and Parameter Definition

This section prepares the core dataset and defines all key parameters for subsequent analysis. It ensures data quality, feature consistency, and centralized parameter management.

1.1 Initial Parameter Definitions

Purpose: Establish economic and technical constants used throughout the analysis.
Why: Centralizes conversion rates and cost assumptions for consistency.
How: Defines exchange rates, technology-specific costs, and valuation metrics.

Listing 1: Economic and technical parameters

```
# --- Economic Parameters ---
fx <- 0.94895              # USD/EUR exchange rate
c_inv_solar_usd <- 778     # USD/kW (s investment)
c_inv_wind_usd <- 1159     # USD/kW (w investment)
c_om_solar_usd <- 7.36     # USD/kW (s Oper. & Maint. Costs)
c_om_wind_usd <- 29.9      # USD/kW (w Oper. & Maint. Costs)

# Convert to EUR
c_inv_s <- c_inv_solar_usd / fx # EUR/kW Investment cost (s)
c_inv_w <- c_inv_wind_usd  / fx # EUR/kW Investment cost (w)
c_om_s <- c_om_solar_usd   / fx # EUR/kW solar Oper. & Maint. Costs (s)
c_om_w <- c_om_wind_usd    / fx # EUR/kW solar Oper. & Maint. Costs (w)

# --- Technical Parameters ---
omega <- 0.01              # Size adjustment factor
epsilon <- 0               # Technology cost shock. Set to 0
life <- 25                 # Plant lifetime (years)
v_mw <- 400                # Consumer valuation (EUR/MWh)
```

1.2 Dataset Preparation

1.2.1 Splitting Oversized Power Plants: Detailed Methodology

Some projects in the dataset are exceptionally large (for example, the Francisco Pizarro Solar Farm at 553 MW). To ensure that these mega-projects do not distort the analysis, any plant exceeding 50 MW is automatically split into smaller, more manageable units. This is accomplished using the `split_project` function, detailed below.

Function Purpose The function splits a grouped tibble (all rows for a single project) into smaller units if the total capacity exceeds a set threshold (`max_capacity`). This preserves the integrity of statistical analysis by preventing large plants from dominating results.

IMPORTANT — MAKE SURE THAT:

1. `Wind_Solar_projects_Spain_2022 - merged.csv` &
2. `Wind_Solar_projects_with_coordinates_Spain_2022.xlsx`

are located in your raw data directory `data_raw`, please, to be able to run the code smoothly.

Listing 2: Plant splitting function

```r
# Load key datasets
file_name <- "Wind_Solar_projects_Spain_2022 - merged.csv"

file_path <- fs::path(data_raw, file_name)

data_solar_wind_proj_2022 <- read_csv(file_path)

# Some plants are really big, so we decide to split them, in order to
    have better results.

#  Max capacity per plant
max_capacity <- 50

split_project <- function(df_group) {
  total_capacity <- sum(df_group$capacity)
  if (total_capacity <= max_capacity) {
    return(df_group)
  }
  n_splits <- ceiling(total_capacity / max_capacity)
  cap_split <- rep(floor(total_capacity / n_splits * 1000) / 1000, n_
     splits)
  cap_split[length(cap_split)] <- round(total_capacity - sum(cap_split
     [-length(cap_split)]), 3)
  tibble(
    projectname = str_c(unique(df_group$projectname), " ", seq_len(n_
       splits)),
    capacity = cap_split,
    avgcapacityfactor = rep(df_group$avgcapacityfactor[1], n_splits),
    type = rep(df_group$type[1], n_splits)
  )
}
```

Step-by-Step Explanation

1. **Calculate total capacity:** Sums the capacity of all units in the group.
2. **Check if splitting is needed:** If the total is less than or equal to `max_capacity`, returns the group unchanged.
3. **Determine number of splits:** Uses the ceiling function to ensure all new units are below or equal to the threshold.
4. **Assign capacities:** All but the last unit get the same base capacity (rounded down to 3 decimals). The last unit receives the remaining capacity to ensure the sum is exact.
5. **Create new records:** Returns a tibble with unique names for each split (e.g., "Francisco Pizarro 1", "Francisco Pizarro 2", "Francisco Pizarro 3", *etc.*), and replicates the original type and capacity factor.

Key Features

- **Precision:** Maintains 3-decimal accuracy, ensuring the sum of splits equals the original capacity.
- **Attribute Preservation:** Keeps original project type and capacity factor for all splits.

- **Unique Naming:** Each split is uniquely named for traceability.
- **Error Prevention:** The rounding adjustment for the last unit ensures no loss or inflation of total capacity.

Example: Francisco Pizarro Solar Plant

Output:

projectname	capacity	type
Francisco Pizarro 1	46.083	Solar
Francisco Pizarro 2	46.083	Solar
⋮	⋮	⋮
Francisco Pizarro 11	46.083	Solar
Francisco Pizarro 12	46.087	Solar

Total: $11 \times 46.083 + 46.087 = 553.00$ MW

This approach ensures that all plants, regardless of their original size, contribute fairly to the statistical analysis, and that cost and performance metrics are not biased by the presence of outliers.

1.2.2 Applying the Splitting Function to the Dataset

After defining the `split_project` function, we need to apply it to the entire dataset so that all oversized plants are properly split into smaller, manageable units. This is accomplished with the following code:

Listing 3: Applying the splitting function to all projects

```
data_solar_wind_proj_2022 <- data_solar_wind_proj_2022 |>
  group_by(projectname) |>
  group_split() |>
  map_dfr(split_project) |>
  ungroup()
```

Step-by-Step Explanation:

1. **Group by Project Name:** The dataset is grouped by `projectname` so that each group contains all rows corresponding to a single project.
2. **Split into List of Groups:** The `group_split()` function converts the grouped data into a list, where each element is a tibble for one project.
3. **Apply Splitting Function:** The `map_dfr(split_project)` function applies the `split_project` function to each group (i.e., each project). This returns a new tibble for each project, with oversized ones split as needed.
4. **Combine Results:** The results for all projects are combined into a single data frame (using `map_dfr`, which row-binds the results).
5. **Remove Grouping:** The `ungroup()` function ensures the final dataset is not grouped, restoring it to a regular tibble.

The `map_dfr` function from the `purrr` package is a powerful and useful tool for applying a function to each element of a list (or vector) and then combining the results into a single data frame by row-binding them together. In our workflow, after splitting the dataset into a list of project groups, we use `map_dfr` to apply the `split_project` function to each group.

Outcome: After this process, `data_solar_wind_proj_2022` contains only plants that do not exceed the maximum allowed capacity (here: 50 MW). Each original project is now represented by one or more rows, each corresponding to a split unit if necessary, with all relevant attributes (such as type and capacity factor) preserved. This prepares the dataset for robust, unbiased analysis in subsequent steps.

1.3 Feature Engineering and Key Analysis Parameters

After splitting oversized plants, we further process the dataset to compute essential features and set up key parameters for the subsequent simulations.

Feature Engineering The following code block creates new variables for each project or project segment, enabling robust comparison and downstream modeling:

Listing 4: Feature engineering and cost calculation

```
wind_solar_proj_2022 <- data_solar_wind_proj_2022 |>
  mutate(
    hours = (avgcapacityfactor * 8760),        # Total hours worked
                                               # in a year

    power_kw = capacity * 1000,                # Capacity (kW)

    q_i_kwh = hours * life * power_kw,         # Total production (kWh)

    q_i_mwh = hours * life * capacity,         # Total production (MWh)

    v_q_i_mwh = v_mw * q_i_mwh                 # (EUR/MWh)*MWh = EUR
) |>
  mutate(
    c_inv = case_when(
      type == "Solar" ~ c_inv_s,
      type == "Wind"  ~ c_inv_w
    ),
    c_om = case_when(
      type == "Solar" ~ c_om_s,
      type == "Wind"  ~ c_om_w
    ),
    total_cost = (c_inv + c_om * life + epsilon) * (power_kw + (1000 -
       power_kw) * omega),
    avg_cost_euro_kwh = (total_cost / q_i_kwh),
    avg_cost_euro_mwh = (total_cost / q_i_mwh)
  ) |>
  arrange(projectname, capacity)
```

Explanation:

- `hours`: Calculates the total number of hours the plant operates in a year, based on its average capacity factor.
- `power_kw`: Converts plant capacity from MW to kW for cost calculations.
- `q_i_kwh`, `q_i_mwh`: Compute the total energy production over the plant's lifetime in both kWh and MWh.
- `v_q_i_mwh`: Calculates the total economic value of the plant's output, using a fixed consumer valuation.
- `c_inv`, `c_om`: Assign investment and O&M costs based on technology type.
- `total_cost`: Computes the total cost over the plant's life, including a small adjustment for projects (ω) and any technology cost shock (ϵ).
- `avg_cost_euro_kwh`, `avg_cost_euro_mwh`: Calculate the average cost per unit of energy produced in kWh and MWh, respectively.

Key Parameters for Simulation The next code block defines all core parameters required for economic simulation and scenario analysis. These include price scaling, probability distribution parameters, scenario vectors, and policy levers.

Listing 5: Key simulation and scenario parameters

```
# Key Parameters for Subsequent Sections
x <- 60                    # Price scaling factor

alpha <- 4                 # Beta distribution parameter (alpha)
beta <- 2                  # Beta distribution parameter (beta)
expected_p <- (alpha / (alpha + beta)) # Expected value of p
var_p <- (alpha*beta) / ((alpha+beta)^2*(alpha+beta+1)) # Variance of p

threshold_price <- x * expected_p          # Price threshold for
                                           # contracts
                                           # threshold is 60(2/3) = 40


gamma_values <- seq(0, 0.5, by = 0.025)    # Share of opportunistic
                                           # buyers

r_0 <- 1.334653e-07                        # Arbitrary reference value
                                           # (should be small enough)

lambda <- 0.3                              # Per-unit cost of soc. funds

# Contract demand scenarios
theta_no_cpr <- 3500                       # No-CPR scenario
theta_values <- c(2500, 3500, 4500)        # CPR scenarios
                                           # (those will become clear
                                              afterwards)

# For public subsidies section
selected_gammas <- seq(from = 0, to = 0.5, by = 0.1)
selected_theta <- 3500

# Regulator-backed contracts section
total_contract_demand <- 5000

private_demand <- 2500

rbc_demand <- 2500

theta_comparison_rbc <- 2500

# Plotting font size

base_s <- 25

## Color Palette ----------------------
# Assign base palette based on selected theme

base_palette <- switch(
  selected_color_theme,
  "Green" = green_palette_base,
  "Blue" = blue_palette_base,
  stop("Unknown color theme. Choose 'Green' or 'Blue', please.")
)
```

```
theme_palette_gamma <- gradient_n_pal(base_palette)(seq(0, 1, length.
    out = length(gamma_values)))
theme_palette_theta <- gradient_n_pal(base_palette)(seq(0, 1, length.
    out = length(theta_values)))
theme_palette_welfare <- gradient_n_pal(base_palette)(seq(0, 1, length.
    out = 3))
theme_palette_avg_cost_graphs <- base_palette[2]
theme_palette_map <- c("Wind" = base_palette[1], "Solar" = base_palette
    [3])
```

Explanation:

- `x`, `alpha`, `beta`: Control the price and distribution of contract adoption probabilities.
- `gamma_values`, `theta_values`: Define scenario grids for policy simulations (e.g., varying the share of opportunistic buyers and contract demand).
- `lambda`: Used for modeling the cost of public funds in policy scenarios.
- `selected_gammas`, `selected_theta`: Used to select a subset of scenarios for clearer figures.
- `total_contract_demand`, `private_demand`, `rbc_demand`: Parameters for regulator-backed contracts.
- `base_s`: Standardizes the font size for all plots.

Color Palette and Plotting Themes Finally, the code block allows to select a consistent color palette for all plots, with a single line change. This ensures that all visualizations are either presentation-friendly (green) or publication-ready (blue).
Explanation:

- `base_palette`: Sets the base color scheme for all subsequent plots.
- `theme_palette_*`: Automatically generates color gradients for scenario plots, ensuring visual consistency and clarity.