Specialisation Project (VT2)
HS2024

Enhanced Platform for Investment Analysis

Mixed-Integer Linear Programming Optimization Model for Integrated Energy Systems in Python

Submitted by
Rui Vieira
Institute of Product Development and Production Technologies (IPP)

Supervisor
Dr. Andrea Giovanni Beccuti
IEFE Model Based Process Optimisation

Study Program
Business Engineering, MSc in Engineering

Zurich University of Applied Sciences
School of Engineering

July 7, 2025

Imprint

Project: Specialisation Project (VT2)
Title: Enhanced Platform for Investment Analysis
Author: Rui Vieira
Date: July 7, 2025
Keywords: mixed-integer linear programming, MILP, quantitative modeling, python, strategic planning, optimization, asset valuation, power-flow, platform, forecasting, energy trading

Study program:
Business Engineering, MSc in Engineering
ZHAW School of Engineering

Supervisor:
Dr. Andrea Giovanni Beccuti
IEFE Model Based Process Optimisation
Email: giovanni.beccuti@zhaw.ch

Abstract

This work presents an integrated investment framework for energy systems, focusing on optimal technology selection and placement of electrical generation, conversion, and storage assets. The core engine combines DC Optimal Power Flow (DC-OPF) simulations with linear programming (using the PuLP solver) to evaluate both technical feasibility and economic viability across a multi-scenario analysis. A 9-bus test network provides the backdrop for a reduced ten distinct cases, each featuring a unique mix of conventional (nuclear, gas) and renewable (solar, wind) power plants, supplemented by battery storage of varying capacities.

To balance computational efficiency with seasonal realism, the annual horizon is divided into three representative weeks (summer, winter, and spring/autumn), whose costs and operations are subsequently scaled to form a full-year analysis. This approach reveals significant seasonal differences in storage utilization even enabling clean assets to compensate for the cost of conventional generation.

In scenarios featuring abundant solar generation, the SoC frequently reaches its upper limits, highlighting the potential for upsizing or more flexible operational strategies—such as battery leasing or modular additions—to capture peak renewable output.

An economic sensitivity analysis underscores the strong influence of high-cost resources during extreme load conditions, causing a disproportionate rise in total costs when reliance on expensive generation escalates. Meanwhile, scenarios with nuclear-dominated baseload exhibit lower operational cost volatility but may still benefit from targeted storage deployment to manage residual demand swings. AI-assisted reporting consolidates these findings by identifying cost drivers, optimal technology mixes, and operational bottlenecks across all scenarios. Notably, Scenario7's balanced blend of nuclear, solar, and wind with moderate battery support emerges as the most cost-effective configuration, while Scenario4, featuring gas-fired generation and multiple storage units, proves the least favorable in terms of net present value (NPV).

Overall, the proposed framework bridges technical dispatch simulation and investment analysis, guiding stakeholders in designing resilient, economically viable energy systems. Future enhancements include broader maintenance modeling, real-time price integration for advanced arbitrage strategies, and further prompt-engineering improvements to refine AI-driven reporting and decision support.

Keywords: mixed-integer linear programming, MILP, quantitative modeling, python, strategic planning, optimization, asset valuation, power-flow, platform, forecasting, energy trading

Contents

1 Introduction

1.1 Context & Motivation

1.2 VT1 Recaps

1.2.1 Bottlenecks

1.3 Goals

2 Literature and Toolchain Review

2.1 Mixed-Integer Programming in Power-System Planning

2.2 Short-term PV/Wind Forecasting Methods

2.3 Solver Landscape and Selection

2.4 Python Ecosystem for Optimisation and ML

3 Problem Definition and Scope

3.1 Planning Horizon and System Boundaries

3.2 Decision Variables and Constraints

3.3 Forecast Horizon and Accuracy Targets

3.4 KPIs

4 Methodology

4.1 Data Pipeline and Representative Weeks

A reorganization of the data structure was made, with all relevant grid data now consolidated in the `data/grid` directory. The configuration was also simplified, moving from a more complex setup to a single `analysis.json` file, which streamlines the definition of planning horizons, load growth, and representative weeks.

The grid data is organized in a clear directory structure:

```
data/grid/
|-- analysis.json      # Configuration file
|-- buses.csv          # Bus parameters and coordinates
|-- generators.csv     # Gen. spec (type, lifetime, capex, etc.)
|-- lines.csv          # Transmission line
|-- loads.csv          # Load nodes and coordinates
`-- storages.csv       # Storage spec. (lifetime, capex, etc.)
```

Each CSV file contains the essential parameters for power system modeling. For example, `generators.csv` includes capacity, costs, and technical parameters for each generator; `storages.csv` contains energy capacity, power rating, and efficiency values; while `loads.csv` defines consumption points. This structure enables straightforward data updates and maintains clear separation of concerns between different grid components.

The forecasting component, developed in the second part of the semester, further advanced the pipeline. While the initial PV dataset covered only one year with a simple two-column format (`time,value`), the updated approach extended the dataset to over ten years and incorporated a richer set of predictors, including meteorological variables such as temperature, precipitation, cloud cover, and various irradiance measures. This data was fetched and processed using scripts like `forecast0/pv-renewables.py`, reflecting a more sophisticated feature engineering process to enhance forecasting accuracy. The configuration for this stage is managed via a dedicated `config.yaml` file, highlighting the evolution towards a more modular and robust pipeline. These improvements demonstrate a growing understanding of data structure and pipeline design, though further refinements are still possible to optimize efficiency and maintainability.

4.2 Linear to Mixed-Integer Programming Transition

This section describes the transition from a single-year linear program (LP) to a multi-year mixed-integer linear program (MILP) that embeds investment decisions directly in the optimisation. The migration affects five aspects: the mathematical formulation, capital cost treatment, binary variable design, solver configuration, and the solver back-end. Figure 4-2 gives an overview; each element is detailed below.

4.2.1 Formulation (Mathematical Model)

The MILP extends the LP by introducing planning years $y \in Y$, representative seasons $\sigma \in \Sigma$, and binary investment variables. Key sets: g (generators), s (storage), b (buses), l (lines), y (years), t (hours), σ (seasons). Decision variables include: $\mathsf{build}_{g,y}$, $\mathsf{build}^{\text{stor}}_{s,y} \in \{0,1\}$ (commissioning), $\mathsf{inst}_{g,y}$, $\mathsf{inst}^{\text{stor}}_{s,y} \in \{0,1\}$ (availability), and operational variables for each (σ, y, t). The objective minimises operational cost and annualised CAPEX:

$$\min \underbrace{\sum_{y\in Y}\sum_{\sigma\in\Sigma} w_\sigma \sum_{t\in T}\sum_{g\in G} c_g^{\text{marg}}\, p_{g,\sigma,y,t}}_{\text{operational cost}} + \underbrace{\sum_{y\in Y}\left(\sum_{g\in G} A_g\, \mathsf{inst}_{g,y} + \sum_{s\in S} A_s\, \mathsf{inst}^{\text{stor}}_{s,y}\right)}_{\text{annualised CAPEX}}$$

Constraints include: (1) capacity limits, (2) nodal balance, (3) storage dynamics, and (4) binary linking for asset lifetime. The binary linking ensures at most one build per rolling lifetime window, enforcing realistic replacement logic.

4.2.2 Lifetime and Annuity CAPEX Handling

Instead of a lump-sum CAPEX, the MILP internalises an annuity that spreads capital and fixed OPEX over the asset's life. The discount series $D(L,i)$, net-present value NPV_a, and capital-recovery factor $\text{CRF}(L,i)$ yield the annualised cost A_a used in the objective. These are computed once per asset and multiplied by the installed-status binaries, eliminating further discounting in the objective. Implementation is in `optimization.py` (generators: lines 330–360; storage: 388–413).

4.2.3 Chunk-based Binary Installation Variables

The classical year-by-year stock balance scales poorly. The chunk formulation collapses this to one binary per asset-year: $\mathsf{build}_{a,y} = 1$ activates a whole lifetime chunk, and overlapping chunks are forbidden. This reduces the number of binaries and constraints, eliminates slack variables, and natively supports retirement. Implementation is compact (lines 96–108, 104–118). Benefits: constant binaries per asset, no duplicates, and no load-shedding slack.

4.2.4 Branch-and-cut Improvements

Switching to CPLEX enables advanced MIP heuristics: a strict time limit (18 min), parallel threads (10), and tight MIP gap tolerances (1% relative, 1.0 absolute). CPLEX's automatic lazy-cut generation accelerates convergence—branch counts dropped from > 100k (CBC) to ≈ 3k. The problem remains convex except for binaries, so these features yield substantial speed-ups.

4.2.5 Solver Change (Gurobi/GLPK $\to$ CPLEX)

The API moved from PuLP (CBC, GLPK) to CVXPY 2.0+ (CPLEX). The paradigm shifted from single-scenario LP to multi-year MILP. Academic CPLEX is used via a thin adapter: `investment_multi()` wraps the model, calls `prob.solve(solver=cp.CPLEX, ...)`, and serialises results. Vectorised constraint building (lines 142–206) replaces nested Python loops, leveraging CVXPY's broadcasting for a 7× speed-up in model build time.

Recap: Migrating from LP to MILP enabled discrete investment timing, lifetime-based replacement, and a realistic annuity cost model, while keeping operating constraints linear. Combined with tighter branch-and-cut controls and a high-performance solver, the new formulation solves 10-year planning cases in under 20 minutes, versus several hours previously.

4.3 Forecasting Module

4.3.1 Feature Engineering

4.3.2 Model Pool

4.3.3 Hyperparameter Tuning

4.3.4 Error Propagation Scenarios for MILP

4.4 Architecture

5 Implementation

5.1 Code Walk-through

5.2 Data Structures & File Formats

5.3 Performance Profiling and Optimisation

5.4 Testing & Validation Strategy

6 Results

6.1 Forecasting Accuracy

6.2 MILP vs Legacy LP

6.3 Solver Impact (CPLEX vs GLPK)

6.4 Sensitivity and Scenario Analysis

6.5 Integrated Workflow Demo

7 Discussion

7.1 Interpretation of Key Findings

7.2 Trade-offs Analysis

7.3 Limitations

8 Conclusions and Outlook

8.1 Achievements Relative to Goals

8.2 Near-term Tasks

8.3 Long-term Vision

Acknowledgements

For the redaction of this report, I would like to acknowledge the use of artificial intelligence to improve the clarity and structure of my sentences. The core observations, analyses, and personal reflections are entirely my own, drawn from my experiences during the field trip and subsequent research. The LLM usage was employed primarily for language refinement, code formatting, and orthographic corrections. Its integration helped communicate complex concepts clearly and effectively.

A Data Files & Directory Layout

B Major Code Listings

B.1 Optimization Model

```
#!/usr/bin/env python3
"""
Optimization module for multi-year DC OPF with annualized (annuity) investments.
No layering: create, solve, extract numeric results, store in integrated_network.
"""

import cvxpy as cp
import pandas as pd
import numpy as np
import os
import logging

# Configure only a file handler, no console handler
logger = logging.getLogger(__name__)
# Don't add any handlers here - we'll configure through the main module

def compute_crf(lifetime, discount_rate):
    """
    Compute the capital recovery factor (CRF) for an asset.

    CRF = i * (1+i)^n / ((1+i)^n - 1)
    where:
      i = discount_rate,
      n = lifetime (years)

    Returns 1.0 if lifetime is zero or close to zero.
    """
    if lifetime is None or lifetime <= 0:
        return 1.0
    i = discount_rate
    n = lifetime
    numerator = i * (1 + i)**n
    denominator = (1 + i)**n - 1
    if abs(denominator) < 1e-9:
        return 1.0
    return numerator / denominator

def compute_discount_sum(lifetime, discount_rate):
    """
    Compute the sum of discounted factors over the asset's lifetime:

    A = sum_{t=1}^{n} 1/(1+i)^t = (1 - 1/(1+i)^n) / i  if i>0
    """
    if discount_rate <= 0:
        return lifetime
    return (1 - 1 / ((1 + discount_rate) ** lifetime)) / discount_rate

def dcopf(integrated_network):
    """
    Create an integrated multi-year DC OPF problem with:
      - Separate 'build' and 'installed' binary variables per asset & year.
      - Generator dispatch & storage variables for each year+season.
```

```
        - Summed operational + annualized capital cost objective (cost based on 'build').
        - Lifetimes handled by linking 'build' to 'installed' status.

    Returns:
      problem_dict = {
        'objective': cp.Minimize(...),
        'constraints': [...],
        'variables': {
          'gen_build': {...},         # New build variables
          'storage_build': {...},   # New build variables
          'gen_installed': {...},     # Operational status
          'storage_installed': {...},# Operational status
          'season_variables': {season: {...}}
        },
        'years': [...],
        'seasons': [...],
      }
    """
    years = integrated_network.years
    seasons = list(integrated_network.season_networks.keys())
    first_network = list(integrated_network.season_networks.values())[0] if seasons else None

    generators = first_network.generators.index if first_network else []
    storage_units = first_network.storage_units.index if first_network else []
    buses = first_network.buses.index if first_network else []
    lines = first_network.lines.index if first_network else []

    # -- 1) Create global 'build' and 'installed' variables
    gen_build = {(g, y): cp.Variable(boolean=True, name=f"gen_build_{g}_{y}") for g in generators for y in years}
    storage_build = {(s, y): cp.Variable(boolean=True, name=f"storage_build_{s}_{y}") for s in storage_units for y in years}

    gen_installed = {(g, y): cp.Variable(boolean=True, name=f"gen_installed_{g}_{y}") for g in generators for y in years}
    storage_installed = {(s, y): cp.Variable(boolean=True, name=f"storage_installed_{s}_{y}") for s in storage_units for y in years}

    # -- 2) Link 'build' to 'installed' status based on lifetime
    global_constraints = []

    # For each generator g, link installed[g, y] to build decisions within its lifetime
    for g in generators:
        lifetime_g = first_network.generators.at[g, 'lifetime_years']
        # Ensure lifetime is treated as an integer for range calculation
        lifetime_g_int = int(lifetime_g) if pd.notna(lifetime_g) else 0

        # ---- NEW: "at-most-one-build-per-lifetime window" ----
        for y_idx, y in enumerate(years):
            window_builds = [gen_build[(g, yb)]
                             for yb_idx, yb in enumerate(years)
                             if (y_idx - yb_idx) < lifetime_g_int and y_idx >= yb_idx]
            global_constraints.append(cp.sum(window_builds) <= 1)

        # ---- Equality definition of installed ----
        for y_idx, y in enumerate(years):
            window_builds = [gen_build[(g, yb)]
                             for yb_idx, yb in enumerate(years)
```

```
                             if (y_idx - yb_idx) < lifetime_g_int and y_idx >= yb_idx]
            global_constraints.append(
                gen_installed[(g, y)] == cp.sum(window_builds))

    # For each storage unit s, link installed[s, y] to build decisions within its lifetime
    for s in storage_units:
        lifetime_s = first_network.storage_units.at[s, 'lifetime_years']
        # Ensure lifetime is treated as an integer
        lifetime_s_int = int(lifetime_s) if pd.notna(lifetime_s) else 0

        # ---- NEW: "at-most-one-build-per-lifetime window" ----
        for y_idx, y in enumerate(years):
            window_builds = [storage_build[(s, yb)]
                             for yb_idx, yb in enumerate(years)
                             if (y_idx - yb_idx) < lifetime_s_int and y_idx >= yb_idx]
            global_constraints.append(cp.sum(window_builds) <= 1)

        # ---- Equality definition of installed ----
        for y_idx, y in enumerate(years):
            window_builds = [storage_build[(s, yb)]
                             for yb_idx, yb in enumerate(years)
                             if (y_idx - yb_idx) < lifetime_s_int and y_idx >= yb_idx]
            global_constraints.append(
                storage_installed[(s, y)] == cp.sum(window_builds))

    # -- 3) Create flat variable dictionaries and constraints
    # ----------------------------------------------------------
    # Flat dictionaries indexed (asset, year, season)
    # ----------------------------------------------------------
    p_gen, p_line = {}, {}
    p_charge, p_discharge, soc = {}, {}, {}

    flat_constraints = []            # replaces season_constraints

    # Build lookup dictionaries to keep loops light
    mc_g = {}  # Marginal costs for generators
    for g in generators:
        if g in first_network.generators.index:
            mc_g[g] = first_network.generators.at[g, 'marginal_cost']

    # Pre-compute load dictionaries and bus connections
    bus_load_dict = {}
    g_at_bus = {}
    s_at_bus = {}
    lines_from = {}
    lines_to = {}

    for season in seasons:
        net = integrated_network.season_networks[season]
        T = net.T

        # Initialize load dictionary for this season
        bus_load_dict[season] = {b: np.zeros(T) for b in buses}
        g_at_bus[season] = {b: [] for b in buses}
        s_at_bus[season] = {b: [] for b in buses}
        lines_from[season] = {b: [] for b in buses}
        lines_to[season] = {b: [] for b in buses}

        # Build load dictionary
```

```
        if not net.loads.empty:
            for ld in net.loads.index:
                load_bus_orig = net.loads.at[ld, 'bus']
                # Try to match bus types (int vs str)
                matched_bus = None
                for b in buses:
                   try:
                       if str(load_bus_orig) == str(b):
                           matched_bus = b
                           break
                   except: # Handle potential type errors during comparison
                        pass

                if matched_bus is not None:
                    if 'p' in net.loads_t and ld in net.loads_t['p']:
                        load_ts = net.loads_t['p'][ld].values[:T]
                        bus_load_dict[season][matched_bus] += load_ts
                    else:
                        static_val = net.loads.at[ld, 'p_mw']
                        if pd.notna(static_val):
                            bus_load_dict[season][matched_bus] += np.ones(T) * static_val
                        else:
                            logger.warning(f"Load {ld} at bus {matched_bus} has invalid p_mw value: {static_val}. Setting to 0.")

        # Build connection dictionaries
        for g in generators:
            if g in net.generators.index:
                bus_g = net.generators.at[g, 'bus']
                for b in buses:
                    if str(bus_g) == str(b):
                        g_at_bus[season][b].append(g)
                        break

        for s in storage_units:
            if s in net.storage_units.index:
                bus_s = net.storage_units.at[s, 'bus']
                for b in buses:
                    if str(bus_s) == str(b):
                        s_at_bus[season][b].append(s)
                        break

        for l in lines:
            if l in net.lines.index:
                from_bus = net.lines.at[l, 'from_bus']
                to_bus = net.lines.at[l, 'to_bus']
                for b in buses:
                    if str(from_bus) == str(b):
                        lines_from[season][b].append(l)
                    if str(to_bus) == str(b):
                        lines_to[season][b].append(l)

    # ----------------------------------------------------------
    # Create variables ONCE for every (season, year) pair
    # ----------------------------------------------------------
    for s in seasons:
        net = integrated_network.season_networks[s]
        T = net.T

        for y in years:
            # ---------- generators ----------
```

```python
            for g in generators:
                var = cp.Variable(T, nonneg=True, name=f"p_gen_{s}_{g}_{y}")
                p_gen[(g, y, s)] = var

            # ---------- lines ----------
            for l in lines:
                var = cp.Variable(T, name=f"p_line_{s}_{l}_{y}")
                p_line[(l, y, s)] = var

            # ---------- storage ----------
            for st in storage_units:
                p_charge[(st, y, s)] = cp.Variable(T, nonneg=True, name=f"p_charge_{s}_{st}_{y}")
                p_discharge[(st, y, s)] = cp.Variable(T, nonneg=True, name=f"p_discharge_{s}_{st}_{y}")
                soc[(st, y, s)] = cp.Variable(T, nonneg=True, name=f"soc_{s}_{st}_{y}")

    # Add capacity constraints for generators
    for (g, y, s), var in p_gen.items():
        net = integrated_network.season_networks[s]
        if g in net.generators.index:
            g_nom = net.generators.at[g, 'p_nom']
            g_type = net.generators.at[g, 'type']

            if g_type in ['wind', 'solar'] and \
               'p_max_pu' in net.generators_t and g in net.generators_t['p_max_pu']:
                prof = net.generators_t['p_max_pu'][g].values[:net.T]
                flat_constraints.append(
                    var <= cp.multiply(prof, g_nom) * gen_installed[(g, y)]
                )
            else:
                flat_constraints.append(
                    var <= g_nom * gen_installed[(g, y)]
                )
        else:
            flat_constraints.append(var <= 0)

    # Add constraints for line flows
    for (l, y, s), var in p_line.items():
        net = integrated_network.season_networks[s]
        if l in net.lines.index:
            line_cap = net.lines.at[l, 's_nom'] if hasattr(net.lines, 's_nom') else 0
            flat_constraints.append(cp.abs(var) <= line_cap)

    # Add constraints for storage
    for (st, y, s), charge_var in p_charge.items():
        discharge_var = p_discharge[(st, y, s)]
        soc_var = soc[(st, y, s)]
        net = integrated_network.season_networks[s]
        T = net.T

        if st in net.storage_units.index:
            s_p_nom = net.storage_units.at[st, 'p_nom']
            s_max_hours = net.storage_units.at[st, 'max_hours']
            eff_in = net.storage_units.at[st, 'efficiency_store']
            eff_out = net.storage_units.at[st, 'efficiency_dispatch']
            s_e_nom = s_p_nom * s_max_hours
        else:
            s_p_nom, s_e_nom, eff_in, eff_out = 0, 0, 1, 1

```

```
        # Capacity constraints using storage_installed
        flat_constraints.append(charge_var <= s_p_nom * storage_installed[(st, y)])
        flat_constraints.append(discharge_var <= s_p_nom * storage_installed[(st, y)])
        flat_constraints.append(soc_var <= s_e_nom * storage_installed[(st, y)])

        # SoC dynamics
        flat_constraints.append(
            soc_var[1:] == soc_var[:-1] + eff_in * charge_var[:-1] - (1.0/eff_out) * discharge_var[:-1]
        )
        flat_constraints.append(soc_var[0] == 0)
        # Relax final SoC constraint
        flat_constraints.append(soc_var[T-1] >= 0)
        flat_constraints.append(soc_var[T-1] <= s_e_nom * storage_installed[(st, y)] * 0.1)

    # Nodal power balance constraints
    load_growth_factors = integrated_network.load_growth if hasattr(integrated_network, 'load_growth') else {y: 1.0 for y in years}
    season_weights = integrated_network.season_weights if hasattr(integrated_network, 'season_weights') else {s: 1.0 for s in seasons}

    for s in seasons:
        net = integrated_network.season_networks[s]
        T = net.T
        for y in years:
            growth = load_growth_factors.get(y, 1.0)
            for b in buses:
                load_vec = cp.Constant(growth * bus_load_dict[s][b])

                gen_sum = cp.sum([p_gen[(g, y, s)] for g in g_at_bus[s][b]]) if g_at_bus[s][b] else 0
                st_net = cp.sum([p_discharge[(st, y, s)] - p_charge[(st, y, s)] for st in s_at_bus[s][b]]) if s_at_bus[s][b] else 0
                flow_out = cp.sum([p_line[(l, y, s)] for l in lines_from[s][b]]) if lines_from[s][b] else 0
                flow_in = cp.sum([p_line[(l, y, s)] for l in lines_to[s][b]]) if lines_to[s][b] else 0

                flat_constraints.append(
                    gen_sum + st_net + flow_in == load_vec + flow_out
                )

    # -- 4) Build the objective function: Operational cost + annualized capital cost
    # Operational cost with season weighting
    operational_obj = 0
    for (g, y, s), var in p_gen.items():
        if g in mc_g:  # Using pre-computed marginal costs
            net = integrated_network.season_networks[s]
            weight = season_weights.get(s, 0)
            mc = mc_g[g]
            # Remove discount factor for operational costs
            operational_obj += weight * mc * cp.sum(var)

    capital_obj = 0
    if first_network:
        # Process generator costs using the annuity approach
```

```
        for g in generators:
            if g not in first_network.generators.index: continue

            # Retrieve fixed parameters from the network data
            capex = first_network.generators.at[g, 'capex']
            lifetime = first_network.generators.at[g, 'lifetime_years']
            discount_rate = first_network.generators.at[g, 'discount_rate']
            # Check if operating_costs exists, otherwise default to 0
            opex_fraction = 0.0
            if 'operating_costs' in first_network.generators.columns:
                opex_fraction = first_network.generators.at[g, 'operating_costs']

            # Default lifetime if needed
            if lifetime is None or pd.isna(lifetime) or lifetime <= 0:
                lifetime = 1

            # Compute the sum of discounted factors over the asset's lifetime
            discount_sum = compute_discount_sum(lifetime, discount_rate)
            # Compute the asset's NPV (here treating CAPEX and operating cost as cash outflows)
            npv = capex + (capex * opex_fraction * discount_sum)
            # Compute the capital recovery factor (CRF)
            crf = compute_crf(lifetime, discount_rate)
            # Annual cost (annuity) for the asset
            annual_asset_cost = npv * crf

            # For every planning year, if the generator is installed, add its full annuity cost (without additional discounting)
            for y in years:
                capital_obj += annual_asset_cost * gen_installed[(g, y)]

        # Process storage costs similarly:
        for s in storage_units:
            if s not in first_network.storage_units.index: continue

            # Retrieve fixed parameters from the network data
            capex = first_network.storage_units.at[s, 'capex']
            lifetime_s = first_network.storage_units.at[s, 'lifetime_years']
            discount_rate_s = first_network.storage_units.at[s, 'discount_rate']
            # Check if operating_costs exists, otherwise default to 0
            opex_fraction_s = 0.0
            if 'operating_costs' in first_network.storage_units.columns:
                opex_fraction_s = first_network.storage_units.at[s, 'operating_costs']

            # Default lifetime if needed
            if lifetime_s is None or pd.isna(lifetime_s) or lifetime_s <= 0:
                lifetime_s = 1

            # Compute the sum of discounted factors over the asset's lifetime
            discount_sum_s = compute_discount_sum(lifetime_s, discount_rate_s)
            # Compute the asset's NPV (here treating CAPEX and operating cost as cash outflows)
            npv_s = capex + (capex * opex_fraction_s * discount_sum_s)
            # Compute the capital recovery factor (CRF)
            crf_s = compute_crf(lifetime_s, discount_rate_s)
            # Annual cost (annuity) for the asset
            annual_asset_cost_s = npv_s * crf_s
```

```python

            # For every planning year, if the storage is installed, add its full annuity cost (without additional discounting)
            for y in years:
                capital_obj += annual_asset_cost_s * storage_installed[(s, y)]

    total_cost = operational_obj + capital_obj
    objective = cp.Minimize(total_cost)

    all_constraints = global_constraints + flat_constraints

    return {
        'objective': objective,
        'constraints': all_constraints,
        'variables': {
            'gen_build': gen_build,             # Return new build vars
            'storage_build': storage_build,     # Return new build vars
            'gen_installed': gen_installed,     # Return installed vars
            'storage_installed': storage_installed, # Return installed vars
            'p_gen': p_gen,
            'p_line': p_line,
            'p_charge': p_charge,
            'p_discharge': p_discharge,
            'soc': soc
        },
        'years': years,
        'seasons': seasons
    }

def investement_multi(integrated_network, solver_options=None):
    """
    Solve the integrated multi-year problem using the build/installed formulation:
      1) Create the problem
      2) Solve with CPLEX
      3) Extract numeric variable values (including build vars)
      4) Store results in integrated_network.integrated_results
    """
    if solver_options is None:
        solver_options = {}

    years = integrated_network.years
    seasons = integrated_network.seasons
    first_network = list(integrated_network.season_networks.values())[0] if seasons else None
    logger.info(f"Solving multi-year investment: {len(seasons)} seasons, {len(years)} years using build/installed formulation.")

    # 1) Create problem
    problem_dict = dcopf(integrated_network)
    prob = cp.Problem(problem_dict['objective'], problem_dict['constraints'])

    # 2) Solve with CPLEX
    cplex_params = {'threads': 10, 'timelimit': 18*60, 'mip.tolerances.mipgap': 0.01}
    cplex_params.update(solver_options)

    try:
        logger.info("Solving with CPLEX...")
        # Increase MIP tolerances for potentially challenging problems
        cplex_params.update({'mip.tolerances.absmipgap': 1.0})
```

```
        prob.solve(solver=cp.CPLEX, verbose=True, cplex_params=cplex_params)
    except Exception as e:
        logger.error(f"Solver failed: {e}")
        # Check if the error message contains DCP details
        if "Problem does not follow DCP rules" in str(e):
            logger.error("DCP Error Details:")
            # (CVXPY often prints DCP errors to stdout/stderr, check console )
            # Extracting specific details programmatically can be complex
            pass
        integrated_network.integrated_results = {
            'status': 'failed',
            'value': None,
            'success': False,
            'variables': {}
        }
        return integrated_network.integrated_results

    status = prob.status
    objective_value = prob.value if status in ("optimal", "optimal_inaccurate") else None

    if status not in ("optimal", "optimal_inaccurate"):
        logger.warning(f"Solve ended with status {status}. Objective value: {objective_value}")
        # Attempt to analyze infeasibility if CPLEX provides details
        # (This requires specific CPLEX API calls not directly available via CVXPY)
        integrated_network.integrated_results = {
            'status': status,
            'value': objective_value,
            'success': False,
            'variables': {}
        }
        return integrated_network.integrated_results
    else:
        logger.info(f"Solve successful with status {status}. Objective value: {objective_value:.2f}")


    # 3) Extract variable values
    result_vars = {}
    var_values = {}
    problem_vars = problem_dict['variables']

    # Helper to safely get variable value
    def get_val(var):
        try:
            return float(var.value) if var.value is not None else 0.0
        except (AttributeError, TypeError, ValueError):
            return 0.0 # Default to 0 if value is invalid or missing

    # Extract build variables
    for k, var in problem_vars['gen_build'].items():
        var_values[k] = get_val(var)
        result_vars[f"gen_build_{k[0]}_{k[1]}"] = var_values[k]
    for k, var in problem_vars['storage_build'].items():
        var_values[k] = get_val(var)
        result_vars[f"storage_build_{k[0]}_{k[1]}"] = var_values[k]

    # Extract installed variables
    for k, var in problem_vars['gen_installed'].items():
        var_values[k] = get_val(var)
```

```
            result_vars[f"gen_installed_{k[0]}_{k[1]}"] = var_values[k]
        for k, var in problem_vars['storage_installed'].items():
            var_values[k] = get_val(var)
            result_vars[f"storage_installed_{k[0]}_{k[1]}"] = var_values[k]

        # Extract seasonal dispatch variables using the flat dictionaries
        for var_type, var_dict in [('p_gen', problem_vars['p_gen']),
                                   ('p_line', problem_vars['p_line']),
                                   ('p_charge', problem_vars['p_charge']),
                                   ('p_discharge', problem_vars['p_discharge']),
                                   ('soc', problem_vars['soc'])]:
            for key, var_vec in var_dict.items():
                # key is (asset_id, year, season)
                asset_id, year, season = key
                # Extract values safely
                try:
                    vals = var_vec.value
                    if vals is None:
                        net = integrated_network.season_networks[season]
                        T = net.T
                        vals = np.zeros(T)
                    elif not isinstance(vals, np.ndarray):
                        net = integrated_network.season_networks[season]
                        T = net.T
                        vals = np.full(T, float(vals))
                except (AttributeError, TypeError, ValueError):
                    net = integrated_network.season_networks[season]
                    T = net.T
                    logger.warning(f"Could not get value for {var_type} {key}. Defaulting to zeros.")
                    vals = np.zeros(T)

                # Store individual time step values with padded hour format
                for t in range(len(vals)):
                    result_vars[f"{var_type}_{season}_{asset_id}_{year}_{t+1:03d}"] = float(vals[t])

        # 4) Save solution
        integrated_network.integrated_results = {
            'status': status,
            'value': objective_value,
            'success': True,
            'variables': result_vars
        }

        # Log summary using extracted values from result_vars
        num_gen_built = sum(1 for k, v in result_vars.items() if k.startswith('gen_build_') and v > 0.5)
        num_stor_built = sum(1 for k, v in result_vars.items() if k.startswith('storage_build_') and v > 0.5)
        logger.info(f"Total generator build decisions: {num_gen_built}")
        logger.info(f"Total storage build decisions: {num_stor_built}")

        # Detailed installation analysis using 'installed' variables
        logger.info("Analyzing installation patterns (based on 'installed' status):")
        gen_installations = {}
        storage_installations = {}

        # Use result_vars which contains the extracted float values
        for k, v in result_vars.items():
            if v > 0.5: # Use a threshold for binary check
                if k.startswith('gen_installed_'):
```

```
                parts = k.split('_')
                if len(parts) == 4: # Check format: gen_installed_ID_YEAR
                   try:
                       g = parts[2]
                       y = int(parts[3])
                       if g not in gen_installations: gen_installations[g] = []
                       gen_installations[g].append(y)
                   except (IndexError, ValueError):
                       logger.warning(f"Could not parse key: {k}")
            elif k.startswith('storage_installed_'):
                parts = k.split('_')
                if len(parts) == 4:
                    try:
                        s = parts[2]
                        y = int(parts[3])
                        if s not in storage_installations: storage_installations[s] = []
                        storage_installations[s].append(y)
                    except (IndexError, ValueError):
                        logger.warning(f"Could not parse key: {k}")

    # Function to get lifetime safely
    def get_lifetime(asset_id, asset_type):
        df = first_network.generators if asset_type == 'gen' else first_network.storage_units
        try:
            # Handle potential integer/string mismatch for index lookup
            if asset_id in df.index:
                return df.at[asset_id, 'lifetime_years']
            elif str(asset_id) in df.index:
                 return df.at[str(asset_id), 'lifetime_years']
            elif int(asset_id) in df.index:
                 return df.at[int(asset_id), 'lifetime_years']
            else:
                return "unknown"
        except (KeyError, ValueError, TypeError):
            return "unknown"

    logger.info("Generator installation patterns:")
    for g, years_list in gen_installations.items():
        lifetime = get_lifetime(g, 'gen')
        years_str = ", ".join(str(y) for y in sorted(list(set(years_list)))) # Ensure unique and sorted
        logger.info(f"  Generator {g} (lifetime={lifetime}): installed in years [{years_str}]")

    logger.info("Storage installation patterns:")
    for s, years_list in storage_installations.items():
        lifetime = get_lifetime(s, 'storage')
        years_str = ", ".join(str(y) for y in sorted(list(set(years_list)))) # Ensure unique and sorted
        logger.info(f"  Storage {s} (lifetime={lifetime}): installed in years [{years_str}]")

    return integrated_network.integrated_results
```

Listing 1: DCOPF Implementation

B.2 Solver

```python
#!/usr/bin/env python3
"""
Main entry point for the simplified multi-year power grid optimization tool.

Key changes compared to older versions:
  - Single binary variable for each asset per year (no re-install or replacement).
  - No slack variables (no load shedding).
  - Annualized CAPEX cost (capex_per_mw * p_nom / lifetime).
  - No discounting in the cost function (simple sum across years).
  - Storage SoC forced to zero at season start/end, removing cross-season coupling.
"""

import os
import sys
# Ensure the script can find modules in the scripts directory
sys.path.append(os.path.dirname(os.path.abspath(__file__)))

import json
import argparse
import pandas as pd
from datetime import datetime
import numpy as np
import logging
import traceback

# Import our revised pre-processing, optimization, and post modules
from pre import process_data_for_optimization
from optimization import investement_multi
from post import generate_implementation_plan, plot_seasonal_profiles, plot_annual_load_growth, NumpyEncoder, plot_generation_mix, plot_implementation_timeline
# Import the new production costs analysis module
from analysis.production_costs import analyze_production_costs

# Import your integrated network class, or a similar structure
from network import IntegratedNetwork, Network  # Adjust if your code differs
from components import Bus, Generator, Load, Storage, Branch  # optional if needed

# Season weights are now data-driven, they will be overwritten right after preprocess
SEASON_WEIGHTS = None

def setup_logging(output_dir):
    """Set up logging to both console and file."""
    log_file = os.path.join(output_dir, 'optimization.log')

    # Create a logger
    logger = logging.getLogger('optimization')
    logger.setLevel(logging.INFO)

    # Create handlers
    file_handler = logging.FileHandler(log_file)
    console_handler = logging.StreamHandler()

    # Create formatters
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
    file_handler.setFormatter(formatter)
    console_handler.setFormatter(formatter)
```

```python

    # Add handlers to logger
    logger.addHandler(file_handler)
    logger.addHandler(console_handler)

    return logger

def main():
    """Run the simplified multi-year power grid optimization tool."""
    parser = argparse.ArgumentParser(
        description='Simplified Multi-Year Power Grid Optimization'
    )

    # Input files
    parser.add_argument('--grid-file', required=True,
                        help='Path to grid data directory or CSV file')
    parser.add_argument('--profiles-dir', required=True,
                        help='Directory containing time series profile data')
    parser.add_argument('--analysis-file', required=False, default=None,
                        help='Path to analysis configuration JSON file')

    # Output settings
    parser.add_argument('--output-dir', required=False, default='results',
                        help='Directory to save output files (default: results)')
    parser.add_argument('--save-network', action='store_true',
                        help='Save the optimized network to a pickle file')

    # Optional solver options
    parser.add_argument('--solver-options', type=json.loads, default={},
                        help='JSON string with solver options (e.g. \'{"timelimit":3600}\')')

    # Add flag for running cost analysis
    parser.add_argument('--analyze-costs', action='store_true',
                        help='Run detailed production and cost analysis after optimization')

    args = parser.parse_args()

    # Ensure output directory exists
    os.makedirs(args.output_dir, exist_ok=True)

    # Set up logging
    logger = setup_logging(args.output_dir)
    logger.info("Starting optimization run")

    #------------------------------------------------------------------
    # 1) PRE-PROCESSING
    #    Load data and create the dictionary required for optimization
    #------------------------------------------------------------------
    logger.info("Step 1: Preprocessing data...")
    try:
        processed_data = process_data_for_optimization(
            grid_dir=args.grid_file,
            processed_dir=args.profiles_dir,
            planning_years=None  # or override if you want
        )
        # Get season weights from processed data
        SEASON_WEIGHTS = processed_data.get('season_weights',
                        {'winter':13,'summer':13,'spri_autu':26})
        logger.info("Data preprocessing completed.")
```

```python

        # Log the data summary
        logger.info(f"Processed data summary:")
        logger.info(f"  Grid data contains: {list(processed_data['grid_data'].keys())}")
        logger.info(f"  Seasons: {list(processed_data['seasons_profiles'].keys())}")

        buses_df = processed_data['grid_data']['buses']
        gens_df = processed_data['grid_data']['generators']
        loads_df = processed_data['grid_data']['loads']

        logger.info(f"  Buses: {len(buses_df)}")
        logger.info(f"  Generators: {len(gens_df)}")
        logger.info(f"  Generator types: {gens_df['type'].unique().tolist()}")
        logger.info(f"  Total generation capacity: {gens_df['capacity_mw'].sum()} MW")
        logger.info(f"  Loads: {len(loads_df)}")
        logger.info(f"  Total load: {loads_df['p_mw'].sum()} MW")

        logger.info(f"Generators DataFrame columns: {gens_df.columns.tolist()}")
        logger.info(f"Loads DataFrame columns: {loads_df.columns.tolist()}")
        logger.info(f"Storage DataFrame columns: {processed_data['grid_data'].get('storage_units', processed_data['grid_data'].get('storages', pd.DataFrame())).columns.tolist() if not processed_data['grid_data'].get('storage_units', processed_data['grid_data'].get('storages', pd.DataFrame())).empty else 'No storage units'}")

        # Print loads per bus
        loads_by_bus = loads_df.groupby('bus')['p_mw'].sum()
        logger.info("  Loads by bus:")
        for bus, load in loads_by_bus.items():
            logger.info(f"    Bus {bus}: {load} MW")

        # Print generators per bus
        gens_by_bus = gens_df.groupby('bus')['capacity_mw'].sum()
        logger.info("  Generation by bus:")
        for bus, cap in gens_by_bus.items():
            logger.info(f"    Bus {bus}: {cap} MW")

    except Exception as e:
        logger.error(f"Error during data preprocessing: {e}")
        return 1

    # Check that we have the necessary data
    if not processed_data or 'grid_data' not in processed_data or 'seasons_profiles' not in processed_data:
        logger.error("Error: Incomplete processed data.")
        return 1

    #------------------------------------------------------------------
    # 2) CREATE THE INTEGRATED NETWORK
    #    We'll build an IntegratedNetwork from the preprocessed data.
    #------------------------------------------------------------------
    logger.info("Step 2: Building integrated network object...")

    # Extract analysis info (years, discount, etc.)
    analysis = processed_data['grid_data'].get('analysis', {})
    planning_horizon = analysis.get('planning_horizon', {})
    # We assume 'years' is now a list of relative years [1,2,3,...]
```

```
    years = planning_horizon.get('years', [1])  # fallback if missing
    system_discount_rate = planning_horizon.get('system_discount_rate', 0.0)

    # Extract load growth factors from analysis.json
    load_growth = processed_data['grid_data'].get('analysis', {}).get('load_growth', {})
    load_growth_factors = {}

    # Convert string keys to integers for years
    for year_str, factor in load_growth.items():
        if year_str != 'description':  # Skip the description field
            try:
                year = int(year_str)
                load_growth_factors[year] = float(factor)
            except (ValueError, TypeError):
                logger.warning(f"Skipping invalid load growth entry: {year_str}={factor}")

    logger.info(f"Load growth factors: {load_growth_factors}")

    # Create an IntegratedNetwork instance
    integrated_network = IntegratedNetwork(
        seasons=list(processed_data['seasons_profiles'].keys()),
        years=years,
        discount_rate=system_discount_rate,
        season_weights=SEASON_WEIGHTS
    )

    # Add load growth factors to the integrated network
    integrated_network.load_growth = load_growth_factors

    logger.info(f"Created IntegratedNetwork with:")
    logger.info(f"  Years: {years}")
    logger.info(f"  Seasons: {integrated_network.seasons}")
    logger.info(f"  Discount rate: {system_discount_rate}")
    logger.info(f"  Season weights: {SEASON_WEIGHTS}")

    # For each season, build a sub-network
    for season, season_data in processed_data['seasons_profiles'].items():
        logger.info(f"Building network for season: {season}")
        network = Network(name=season)

        # Add buses from grid data
        buses_df = processed_data['grid_data']['buses']
        for idx, row in buses_df.iterrows():
            network.buses.loc[row['id']] = {
                'name': row['name'],
                'v_nom': row.get('v_nom', 1.0)
            }

        # Add lines
        lines_df = processed_data['grid_data']['lines']
        for idx, row in lines_df.iterrows():
            network.lines.loc[row['id']] = {
                'name': row['name'],
                'from_bus': row['bus_from'],
                'to_bus': row['bus_to'],
                'susceptance': row['susceptance'],
                's_nom': row['capacity_mw']
            }

        # Add generators
        gens_df = processed_data['grid_data']['generators']
```

```
        for idx, row in gens_df.iterrows():
            network.generators.loc[row['id']] = {
                'name': row['name'],
                'bus': row['bus'],
                'p_nom': row['capacity_mw'],
                'marginal_cost': row['cost_mwh'],
                'type': row['type'],
                'capex': row['capex'],
                'lifetime_years': row['lifetime_years'],
                'discount_rate': row['discount_rate']
            }

        # Add loads
        loads_df = processed_data['grid_data']['loads']
        for idx, row in loads_df.iterrows():
            network.loads.loc[row['id']] = {
                'name': row['name'],
                'bus': row['bus'],  # Using 'bus' to match the loads.csv column name
                'p_mw': row['p_mw']
            }

        # Add storage - Note: The file is named 'storages.csv' in the data dir
        storage_df = processed_data['grid_data'].get('storage_units', processed_data['grid_data'].get('storages', pd.DataFrame()))
        for idx, row in storage_df.iterrows():
            network.storage_units.loc[row['id']] = {
                'name': row['name'],
                'bus': row['bus'],
                'p_nom': row['p_mw'],
                'efficiency_store': row['efficiency_store'],
                'efficiency_dispatch': row['efficiency_dispatch'],
                'max_hours': (row['energy_mwh'] / row['p_mw']) if row['p_mw'] else 0,
                'capex': row['capex'],
                'lifetime_years': row['lifetime_years'],
                'discount_rate': row['discount_rate']
            }

        # Set up time snapshots for this season
        T_hours = season_data['hours']  # e.g. 168 for one week
        network.create_snapshots(start_time="2023-01-01", periods=T_hours, freq='h')
        logger.info(f"Created {T_hours} snapshots for season {season}")

        # If there's a generator or load time series, incorporate them
        if 'loads' in season_data:
            loads_ts = season_data['loads']  # multi-index: (time, load_id)
            # Get unique load IDs from the timeseries
            unique_load_ids = loads_ts.index.levels[1]
            logger.info(f"Adding load timeseries for {len(unique_load_ids)} loads in season {season}")
            for load_id in unique_load_ids:
                # Grab the timeseries for this load
                series_mask = loads_ts.xs(load_id, level='load_id')['p_pu']
                # Convert p_pu * nominal
                try:
                    p_nominal = network.loads.loc[load_id, 'p_mw']
                    # Apply the time-varying factor to the nominal load
                    p_series = series_mask.values * p_nominal
                    logger.info(f"Load {load_id}: nominal={p_nominal} MW, min factor={series_mask.min():.2f}, max factor={
```

```python
series_mask.max():.2f}")
                    network.add_load_time_series(load_id, p_series)
                except KeyError:
                    logger.warning(f"Failed to add load timeseries for load {load_id} - not found in network loads")
                    pass  # if mismatch

        # Process generator profiles for wind and solar
        if 'generators' in season_data:
            gens_ts = season_data['generators']
            if not gens_ts.empty:
                # Get unique generator IDs from the timeseries
                unique_gen_ids = gens_ts.index.levels[1]

                # Count how many generators of each type we're adding profiles for
                gen_types = {}
                for gen_id in unique_gen_ids:
                    if gen_id in network.generators.index:
                        gen_type = network.generators.at[gen_id, 'type']
                        gen_types[gen_type] = gen_types.get(gen_type, 0) + 1

                logger.info(f"Adding generator profiles for {len(unique_gen_ids)} generators in season {season}")
                logger.info(f"Generator types with profiles: {gen_types}")

                for gen_id in unique_gen_ids:
                    try:
                        # Check if this generator exists in the network
                        if gen_id not in network.generators.index:
                            logger.warning(f"Generator {gen_id} not found in network generators, skipping")
                            continue

                        # Get the generator type
                        gen_type = network.generators.at[gen_id, 'type']

                        # Only add profiles for wind and solar generators
                        if gen_type in ['wind', 'solar']:
                            # Grab the availability profile (already in MW)
                            p_max_values = gens_ts.xs(gen_id, level='gen_id')['p_max_pu'].values

                            # Add the profile to the network
                            network.add_generator_time_series(gen_id, p_max_values)
                            logger.info(f"Added profile for {gen_type} generator {gen_id} in season {season}")
                        else:
                            logger.info(f"Skipping profile for thermal generator {gen_id}")
                    except Exception as e:
                        logger.warning(f"Failed to add generator profile for {gen_id}: {e}")
                        logger.warning(traceback.format_exc())

        # Add the sub-network to the integrated structure
        integrated_network.add_season_network(season, network)

    #------------------------------------------------------------------
    # 3) OPTIMIZATION
    #    Solve the multi-year investment problem (new approach).
    #------------------------------------------------------------------
```

```
    logger.info("\nStep 3: Solving the multi-year model...")
    try:
        # Log solver options
        logger.info(f"Solver options: {args.solver_options}")

        result = investement_multi(integrated_network, solver_options=args.solver_options)
        if not result or result.get('status') not in ('optimal', 'optimal_inaccurate'):
            logger.error(f"Optimization failed or not optimal. Status: {result.get('status', 'unknown')}")
            return 1
        else:
            logger.info(f"Optimization completed successfully. Objective value: {result.get('value', 0):.2f}")

            # Print a summary of installation decisions
            if 'variables' in result:
                gen_installed = result['variables'].get('gen_installed', {})
                storage_installed = result['variables'].get('storage_installed', {})

                # Check if any generators or storage are selected
                gen_selected = [(g, y) for (g, y), val in gen_installed.items() if val > 0.5]
                storage_selected = [(s, y) for (s, y), val in storage_installed.items() if val > 0.5]

                logger.info(f"Generators installed: {len(gen_selected)}")
                logger.info(f"Storage units installed: {len(storage_selected)}")

                # Print out detailed installation decisions
                logger.info("Detailed generator installation decisions:")
                for (g, y), val in sorted(gen_installed.items()):
                    if val > 0.5:
                        logger.info(f"  Generator {g} installed in year {y}: {val}")

                logger.info("Detailed storage installation decisions:")
                for (s, y), val in sorted(storage_installed.items()):
                    if val > 0.5:
                        logger.info(f"  Storage {s} installed in year {y}: {val}")

                # In a realistic model, some things should be selected. If nothing is selected,
                # the model might need tuning.
                if not gen_selected and not storage_selected:
                    logger.warning("Warning: No generators or storage selected for installation.")
    except Exception as e:
        logger.error(f"Error during optimization: {e}")
        logger.error(traceback.format_exc())
        return 1

    #------------------------------------------------------------------
    # 4) POST-PROCESSING OF RESULTS
    #------------------------------------------------------------------
    logger.info("\nStep 4: Post-processing results...")

    try:
        # Generate implementation plan (even if optimization failed)
```

```
        plan = generate_implementation_plan(integrated_network, args.output_dir)
        logger.info("Implementation plan generated.")
        gen_count = len(plan.get('generators', {}))
        stor_count = len(plan.get('storage', {}))
        logger.info(f"Plan includes {gen_count} generators and {stor_count} storage units")

        # Create visualizations
        timeline_plot = plot_implementation_timeline(integrated_network, args.output_dir)
        logger.info(f"Implementation timeline visualization created: {os.path.basename(timeline_plot) if timeline_plot else 'None'}")

        # Create seasonal profile visualizations
        logger.info("Generating seasonal resource profiles...")
        profile_plots = plot_seasonal_profiles(integrated_network, args.output_dir)
        logger.info("Created profile plots:")
        for season, plot_file in profile_plots.items():
            if plot_file:
                logger.info(f"  - {season}: {os.path.basename(plot_file)}")

        # Plot load growth
        load_growth_plot = plot_annual_load_growth(integrated_network, args.output_dir)
        logger.info(f"Annual load growth visualization created: {os.path.basename(load_growth_plot) if load_growth_plot else 'None'}")

        # Create generation mix plots
        logger.info("Generating generation mix plots...")
        mix_plots = plot_generation_mix(integrated_network, args.output_dir)
        logger.info("Created generation mix plots:")
        for plot_type, plot_file in mix_plots.items():
            if plot_file:
                logger.info(f"  - {plot_type}: {os.path.basename(plot_file)}")

        # Run production and cost analysis if requested
        if args.analyze_costs:
            logger.info("Running detailed production and cost analysis...")
            cost_data = analyze_production_costs(integrated_network, args.output_dir)
            logger.info("Production and cost analysis completed.")

    except Exception as e:
        logger.error(f"Error during post-processing: {e}")
        logger.error(traceback.format_exc())
        return 1

    # Optionally save the network
    if args.save_network:
        integrated_network.save_to_pickle(os.path.join(args.output_dir, 'integrated_network.pkl'))
        logger.info("Optimized integrated network saved to integrated_network.pkl")

    logger.info("\nAll steps completed successfully.")
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Listing 2: Main Solver Implementation

B.3 Forecasting Module

C Mathematical Formulations

C.1 MILP Formulation

C.2 Capital Recovery Factor (CRF) Derivation

C.3 Constraint Sets

D Config/YAML Samples

E Extra Plots & Test Outputs

F Symbols Glossary