Optimization Modeling for Chemical Engineers

Volume 2: Worked Solutions and Engineering Interpretation

Table of contents

Preface

This second volume develops modeling judgment through 18 original teaching problems for graduate students in chemical engineering. Six process themes each progress from a guided model, through an extension, to an open-ended engineering challenge. The data are synthetic and internally defined. These problems are not additional exercises or answer keys from the Williams textbook used in Volume 1.

The central question is not only "What is the optimum?" but also "What assumptions make that answer meaningful?" Students should identify the decision, establish the system boundary and units, predict behavior, solve, and independently check the result.

Learning sequence

Problems	Process theme	New modeling decisions
1-3	Blending	Nominal, box-robust, and budget-robust quality
4-6	Reactors	Throughput, finite operating menus, and static robust operation
7-9	Heat recovery	Dispatch, exchanger selection, and shared design across states
10-12	Batch scheduling	Release times, directed cleaning, and service penalties
13-15	Hydrogen	Inventory, tank sizing, and design with operational recourse
16-18	Water reuse	Dilution, contaminant removal, and treatment selection

Read the data afresh in each problem: the worked instance may change objective terms, information timing, or uncertain parameters. A difference between objectives is meaningful only after those changes are accounted for.

How to work

1. Answer **Before you code** without a solver.
2. Formulate the model with variable domains and units. Draw your process boundary.
3. Implement the model and inspect termination before loading a solution.
4. Complete **Check your solution** using arithmetic or an independent method.
5. Explain one active constraint, one model limitation, and one practical implication.
6. Investigate the extension. For the open-ended levels, defend a recommendation rather than treating the reference policy as the only acceptable answer.

The intended prerequisites are material and energy balances, introductory linear programming, basic integer programming, and Python. Volume 1 provides worked practice with those tools. Each theme can also be used independently because its data are repeated where needed.

Suggested assessment

Allocate 35% to formulation and assumptions, 25% to reproducible implementation, 20% to independent verification, and 20% to interpretation. A numerical match without correct units or information timing is not a complete solution. An alternative feasible design may earn full credit when its objective, scope, and tradeoffs are correctly explained.

Scope of optimality

All reference problems use linear or mixed-integer linear Pyomo models solved by HiGHS. Reactor kinetics are evaluated over a finite menu: optimality applies to that menu, not to a continuous nonlinear reactor design. The water problems assume segregated treatment paths and fixed removal fractions. They explain the bilinear terms in a more general pooling problem, but do not claim to solve that unrestricted problem.

The heat problems use fixed-temperature surrogates. The hydrogen problems assume uniform production and withdrawal within each four-hour period. These assumptions are deliberately visible so that students can challenge them. None of the numerical instances is a validated industrial design.

Worked edition and reproducibility

Each worked chapter follows **Problem Statement, Model Formulation, Pyomo Implementation, Optimal Solution, and Brief Discussion**. Reference solutions are regenerated from the same Python sources that supply the printed model code. There are 18 reproducible Python figures, exported as SVG, embedded-font PDF, and 600 dpi PNG using the lab palette.

Download the complete editable Quarto and Python source. Run `./scripts/build.sh` from its root to rebuild the worked HTML/PDF and the separate student PDF. Setup instructions are in `README.md`. The optimization interpreter is `~/.venvs/optim/bin/python`.

The numerical audit checks active constraints, variable bounds, and integrality at a tolerance of 1e-6 in model units. Additional checks include all robust blend corners, independent reactor-mode enumeration, all batch permutations, heat-network investment patterns, water-treatment alternatives, and physical balances. These checks validate the implementation of the stated model; they do not establish that every process assumption is suitable for a real plant.

Part I

Blending under Uncertainty

1 A nominal blend

Problem 01 | Blending with Uncertain Feed Composition | Level 1: Guided problem

1.1 Problem Statement

A plant blends 100 t/d of a liquid intermediate. Impurity must not exceed 7 wt%. Costs and available flows are deterministic. Impurity has no beneficial role, so only upward deviations matter for the quality upper bound. There are no volume changes, storage, or blending losses.

Feed	Cost (USD/t)	Available (t/d)	Nominal impurity	Maximum upward deviation
A	100	60	0.02	0.01
B	80	60	0.08	0.02
C	55	60	0.14	0.03

All fractions are mass fractions. Deviations of 0.01 mean one percentage point, not a 1% relative change.

Your task. Find the least-cost blend using nominal compositions. Then evaluate that same blend when every impurity takes its upper value. Do not re-optimize during this stress test.

1.1.1 Before you code

Which feed is cheapest? Why can it not supply the entire blend? Will a minimum-cost solution necessarily leave a quality margin?

1.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

1.2 Model Formulation

Let x_i be feed i in t/d, with $0 \le x_i \le 60$. Define c_i as cost, a_i as nominal impurity, and d_i as maximum upward deviation. Let $Q = 100$ t/d and $L = 0.07$.

The mass balance and cost objective are

$$\sum_i x_i = Q, \qquad \min C = \sum_i c_i x_i.$$

The impurity constraint compares impurity mass flow with LQ. Multiplying a fixed quality limit by the fixed product rate avoids a ratio.

At nominal composition, enforce

$$\sum_i a_i x_i \le LQ.$$

This is an LP: all coefficients are known constants. A binding quality constraint means the nominal design has no headroom for upward composition errors.

1.3 Pyomo Implementation

The code below is the model-building portion of `models/blending.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.blending 1
```

```
"""Blend design: nominal, box-robust, and budget-robust quality."""

import itertools
import pyomo.environ as pyo
from models.common import value, solve

COST = {"A": 100, "B": 80, "C": 55}  # USD/t
NOMINAL = {"A": 0.02, "B": 0.08, "C": 0.14}  # mass fraction
DELTA = {"A": 0.01, "B": 0.02, "C": 0.03}
CAP = {"A": 60, "B": 60, "C": 60}  # t/d
TOTAL = 100  # t/d
LIMIT = 0.07  # maximum impurity mass fraction


def build(level, gamma=1.5):
    m = pyo.ConcreteModel(name="Blend quality")
    m.I = pyo.Set(initialize=list(COST))
    m.x = pyo.Var(m.I, domain=pyo.NonNegativeReals, bounds=lambda m, i: (0, CAP[i]))
    m.mass = pyo.Constraint(expr=sum(m.x[i] for i in m.I) == TOTAL)
    nominal = sum(NOMINAL[i] * m.x[i] for i in m.I)
    if level == 1:
        m.quality = pyo.Constraint(expr=nominal <= LIMIT * TOTAL)
    elif level == 2:
        m.quality = pyo.Constraint(
            expr=nominal + sum(DELTA[i] * m.x[i] for i in m.I) <= LIMIT * TOTAL
        )
    else:
        m.p = pyo.Var(domain=pyo.NonNegativeReals)
        m.q = pyo.Var(m.I, domain=pyo.NonNegativeReals)
        m.support = pyo.Constraint(
            m.I, rule=lambda m, i: m.p + m.q[i] >= DELTA[i] * m.x[i]
        )
        m.quality = pyo.Constraint(
            expr=nominal + gamma * m.p + sum(m.q[i] for i in m.I) <= LIMIT * TOTAL
        )
    m.cost = pyo.Objective(expr=sum(COST[i] * m.x[i] for i in m.I))
    m._gamma = gamma
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(1))
```

1.4 Optimal Solution

The reference objective is **8,125.0000 USD/d**. This value is optimal for the explicitly stated model and data.

Feed	Flow (t/d)
A	58.3333
B	0.0000
C	41.6667

Quantity	Value
Nominal impurity (%)	7.0000
Full-box impurity (%)	8.8333
Protected impurity (%)	7.0000
Gamma	0.0000

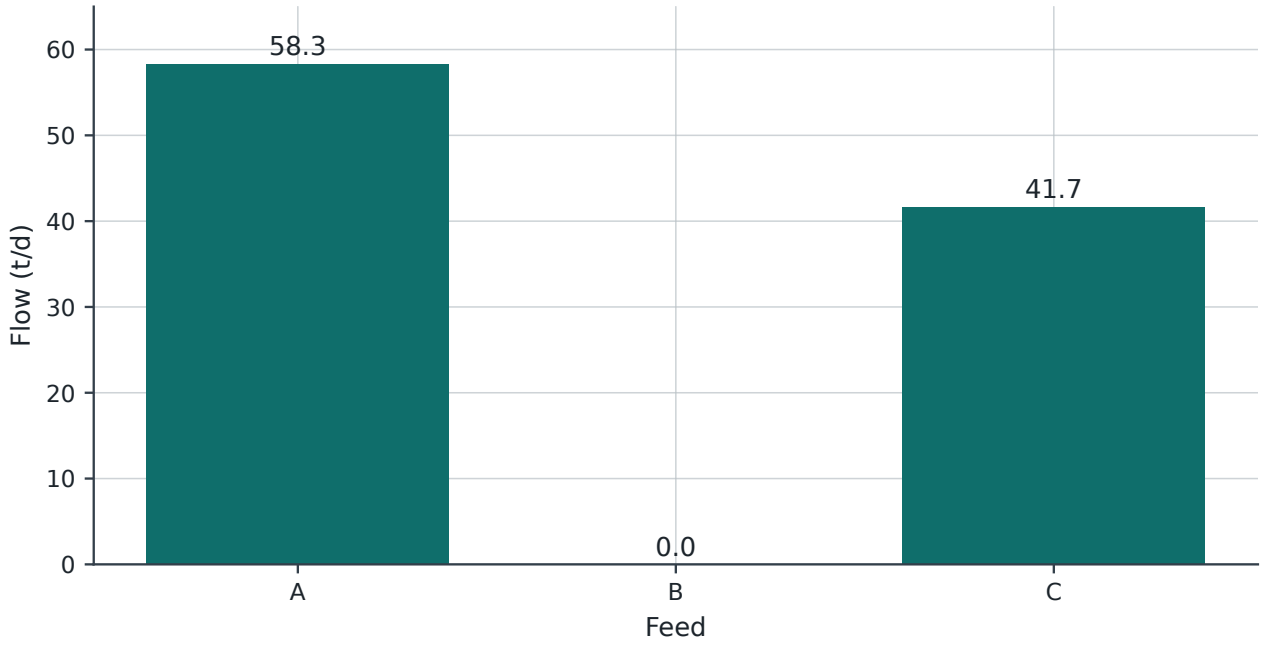

Figure 1.1: Blending with Uncertain Feed Composition: reference solution for level 1.

1.4.1 Check your solution

Independently calculate both total flow and impurity mass flow. Report the stressed composition even though the nominal solver reports an optimal solution.

The automated residual, bound, and integrality audit passed with a maximum violation of 8.88e-16 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

1.5 Brief Discussion

The nominal blend costs 8,125.00 USD/d but reaches 8.833% impurity at the full upper corner. An optimal nominal solution can therefore violate the specification under uncertainty.

1.5.1 Extension to investigate

Predict the effect of reducing feed C availability to 20 t/d, then solve. Explain whether availability or quality now drives the result.

2 A blend protected against all upper deviations

Problem 02 | Blending with Uncertain Feed Composition | Level 2: Model extension

2.1 Problem Statement

A plant blends 100 t/d of a liquid intermediate. Impurity must not exceed 7 wt%. Costs and available flows are deterministic. Impurity has no beneficial role, so only upward deviations matter for the quality upper bound. There are no volume changes, storage, or blending losses.

Feed	Cost (USD/t)	Available (t/d)	Nominal impurity	Maximum upward deviation
A	100	60	0.02	0.01
B	80	60	0.08	0.02
C	55	60	0.14	0.03

All fractions are mass fractions. Deviations of 0.01 mean one percentage point, not a 1% relative change.

Your task. Keep the same production target, prices, and availability. Require the specification to hold for every composition in the box $a_i \le \tilde{a}_i \le a_i + d_i$. Find the robust blend and the cost premium relative to Problem 1.

2.1.1 Before you code

Is testing only the average of the upper and lower compositions enough to guarantee quality? Which box corner is worst when all flows are nonnegative?

2.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

2.2 Model Formulation

Let x_i be feed i in t/d, with $0 \le x_i \le 60$. Define c_i as cost, a_i as nominal impurity, and d_i as maximum upward deviation. Let $Q = 100$ t/d and $L = 0.07$.

The mass balance and cost objective are

$$\sum_i x_i = Q, \qquad \min C = \sum_i c_i x_i.$$

The impurity constraint compares impurity mass flow with LQ. Multiplying a fixed quality limit by the fixed product rate avoids a ratio.

Because $x_i \geq 0$, the maximum impurity occurs at the upper corner:

$$\max_{0 \leq u_i \leq 1} \sum_i (a_i + d_i u_i) x_i = \sum_i (a_i + d_i) x_i.$$

Thus the robust counterpart is simply

$$\sum_i (a_i + d_i) x_i \leq LQ.$$

No probabilities are needed. The guarantee is conditional on the stated bounds being valid.

2.3 Pyomo Implementation

The code below is the model-building portion of `models/blending.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.blending 2
```

```
"""Blend design: nominal, box-robust, and budget-robust quality."""

import itertools
import pyomo.environ as pyo
from models.common import value, solve

COST = {"A": 100, "B": 80, "C": 55}  # USD/t
NOMINAL = {"A": 0.02, "B": 0.08, "C": 0.14}  # mass fraction
DELTA = {"A": 0.01, "B": 0.02, "C": 0.03}
CAP = {"A": 60, "B": 60, "C": 60}  # t/d
TOTAL = 100  # t/d
LIMIT = 0.07  # maximum impurity mass fraction


def build(level, gamma=1.5):
    m = pyo.ConcreteModel(name="Blend quality")
    m.I = pyo.Set(initialize=list(COST))
    m.x = pyo.Var(m.I, domain=pyo.NonNegativeReals, bounds=lambda m, i: (0, CAP[i]))
    m.mass = pyo.Constraint(expr=sum(m.x[i] for i in m.I) == TOTAL)
    nominal = sum(NOMINAL[i] * m.x[i] for i in m.I)
    if level == 1:
        m.quality = pyo.Constraint(expr=nominal <= LIMIT * TOTAL)
    elif level == 2:
        m.quality = pyo.Constraint(
            expr=nominal + sum(DELTA[i] * m.x[i] for i in m.I) <= LIMIT * TOTAL
        )
    else:
        m.p = pyo.Var(domain=pyo.NonNegativeReals)
        m.q = pyo.Var(m.I, domain=pyo.NonNegativeReals)
        m.support = pyo.Constraint(
            m.I, rule=lambda m, i: m.p + m.q[i] >= DELTA[i] * m.x[i]
        )
        m.quality = pyo.Constraint(
            expr=nominal + gamma * m.p + sum(m.q[i] for i in m.I) <= LIMIT * TOTAL
        )
    m.cost = pyo.Objective(expr=sum(COST[i] * m.x[i] for i in m.I))
    m._gamma = gamma
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(2))
```

2.4 Optimal Solution

The reference objective is **8,771.4286 USD/d**. This value is optimal for the explicitly stated model and data.

Feed	Flow (t/d)
A	60.0000
B	22.8571
C	17.1429

Quantity	Value
Nominal impurity (%)	5.4286
Full-box impurity (%)	7.0000
Protected impurity (%)	7.0000
Gamma	3.0000

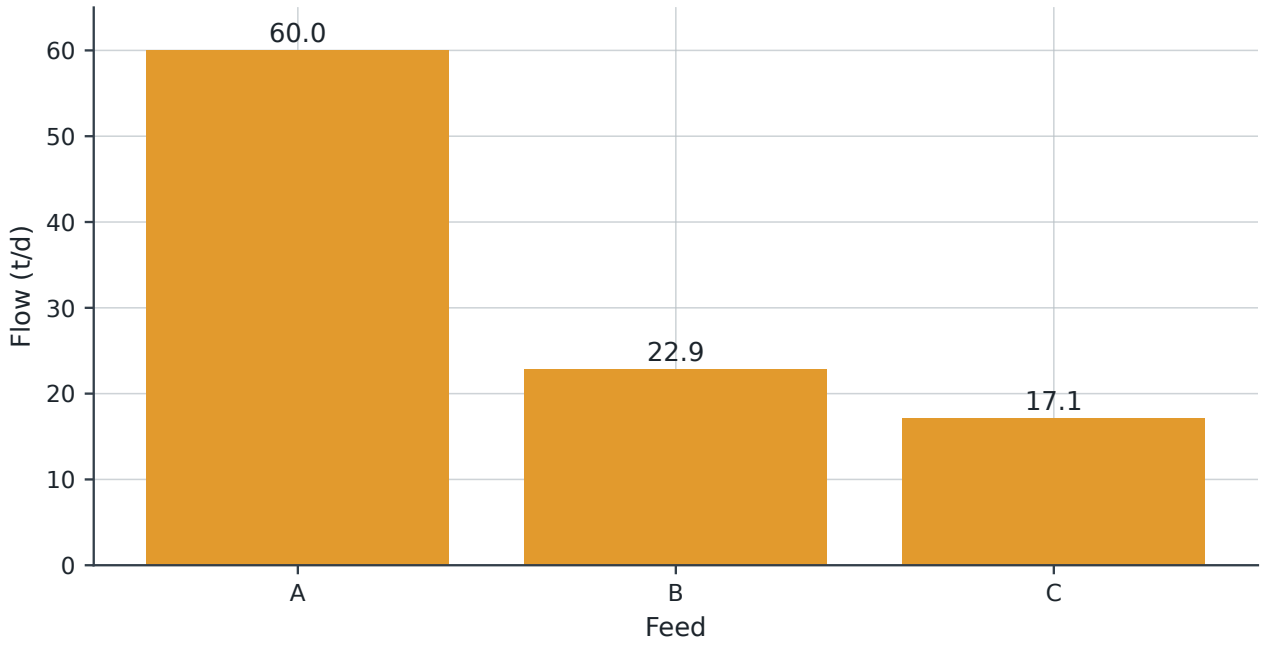

Figure 2.1: Blending with Uncertain Feed Composition: reference solution for level 2.

2.4.1 Check your solution

Check all eight corners of the uncertainty box. Calculate the cost premium as $(C_{box} - C_{nominal})/C_{nominal}$, using the nominal optimum as the denominator.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

2.5 Brief Discussion

The protected design costs 8,771.43 USD/d, a premium of 646.43 USD/d (7.96%) over the nominal optimum. Full-box impurity is 7.000%. Protection at a smaller budget must not be described as full-box protection.

2.5.1 Extension to investigate

Reduce the allowable impurity to 6 wt%. Determine whether the available low-impurity feed still makes the robust model feasible.

3 Choosing an uncertainty budget

Problem 03 | Blending with Uncertain Feed Composition | Level 3: Open-ended challenge

3.1 Problem Statement

A plant blends 100 t/d of a liquid intermediate. Impurity must not exceed 7 wt%. Costs and available flows are deterministic. Impurity has no beneficial role, so only upward deviations matter for the quality upper bound. There are no volume changes, storage, or blending losses.

Feed	Cost (USD/t)	Available (t/d)	Nominal impurity	Maximum upward deviation
A	100	60	0.02	0.01
B	80	60	0.08	0.02
C	55	60	0.14	0.03

All fractions are mass fractions. Deviations of 0.01 mean one percentage point, not a 1% relative change.

Your task. Use $\tilde{a}_i = a_i + d_i u_i$, $0 \le u_i \le 1$, and $\sum_i u_i \le \Gamma$. Solve the reference case $\Gamma = 1.5$, then evaluate $\Gamma = 0, 0.5, 1, 1.5, 2, 2.5, 3$. Recommend a budget and state what evidence would justify it. There is no unique correct recommendation.

3.1.1 Before you code

Does a budget of 1.5 mean that a constraint has a 50% chance of failure? Why should cost be nondecreasing as the budget grows?

3.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

3.2 Model Formulation

Let x_i be feed i in t/d, with $0 \le x_i \le 60$. Define c_i as cost, a_i as nominal impurity, and d_i as maximum upward deviation. Let $Q = 100$ t/d and $L = 0.07$.

The mass balance and cost objective are

$$\sum_i x_i = Q, \qquad \min C = \sum_i c_i x_i.$$

The impurity constraint compares impurity mass flow with LQ. Multiplying a fixed quality limit by the fixed product rate avoids a ratio.

The worst additional impurity is the support function

$$W(x,\Gamma) = \max_{u} \left\{ \sum_i d_i x_i u_i : 0 \le u_i \le 1,\ \sum_i u_i \le \Gamma \right\}.$$

Its LP dual introduces $p \ge 0$ and $q_i \ge 0$, both in t/d of impurity:

$$p + q_i \ge d_i x_i, \qquad \sum_i a_i x_i + \Gamma p + \sum_i q_i \le LQ.$$

Minimizing cost subject to these inequalities is the exact robust counterpart for this uncertainty set. For an independent check, sort $d_i x_i$ from largest to smallest, add the largest $\lfloor \Gamma \rfloor$ terms, and add the fractional part times the next term. $\Gamma = 0$ gives nominal protection; $\Gamma = 3$ gives full-box protection. A budget alone does not establish a probability of failure.

3.3 Pyomo Implementation

The code below is the model-building portion of `models/blending.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.blending 3
```

```
"""Blend design: nominal, box-robust, and budget-robust quality."""

import itertools
import pyomo.environ as pyo
from models.common import value, solve

COST = {"A": 100, "B": 80, "C": 55}  # USD/t
NOMINAL = {"A": 0.02, "B": 0.08, "C": 0.14}  # mass fraction
DELTA = {"A": 0.01, "B": 0.02, "C": 0.03}
CAP = {"A": 60, "B": 60, "C": 60}  # t/d
TOTAL = 100  # t/d
LIMIT = 0.07  # maximum impurity mass fraction


def build(level, gamma=1.5):
    m = pyo.ConcreteModel(name="Blend quality")
    m.I = pyo.Set(initialize=list(COST))
    m.x = pyo.Var(m.I, domain=pyo.NonNegativeReals, bounds=lambda m, i: (0, CAP[i]))
    m.mass = pyo.Constraint(expr=sum(m.x[i] for i in m.I) == TOTAL)
    nominal = sum(NOMINAL[i] * m.x[i] for i in m.I)
    if level == 1:
        m.quality = pyo.Constraint(expr=nominal <= LIMIT * TOTAL)
    elif level == 2:
        m.quality = pyo.Constraint(
            expr=nominal + sum(DELTA[i] * m.x[i] for i in m.I) <= LIMIT * TOTAL
        )
    else:
        m.p = pyo.Var(domain=pyo.NonNegativeReals)
        m.q = pyo.Var(m.I, domain=pyo.NonNegativeReals)
        m.support = pyo.Constraint(
            m.I, rule=lambda m, i: m.p + m.q[i] >= DELTA[i] * m.x[i]
        )
        m.quality = pyo.Constraint(
            expr=nominal + gamma * m.p + sum(m.q[i] for i in m.I) <= LIMIT * TOTAL
        )
    m.cost = pyo.Objective(expr=sum(COST[i] * m.x[i] for i in m.I))
    m._gamma = gamma
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(3))
```

3.4 Optimal Solution

The reference objective is **8,526.3158 USD/d**. This value is optimal for the explicitly stated model and data.

Feed	Flow (t/d)
A	52.6316
B	26.3158
C	21.0526

Quantity	Value
Nominal impurity (%)	6.1053
Full-box impurity (%)	7.7895
Protected impurity (%)	7.0000
Gamma	1.5000

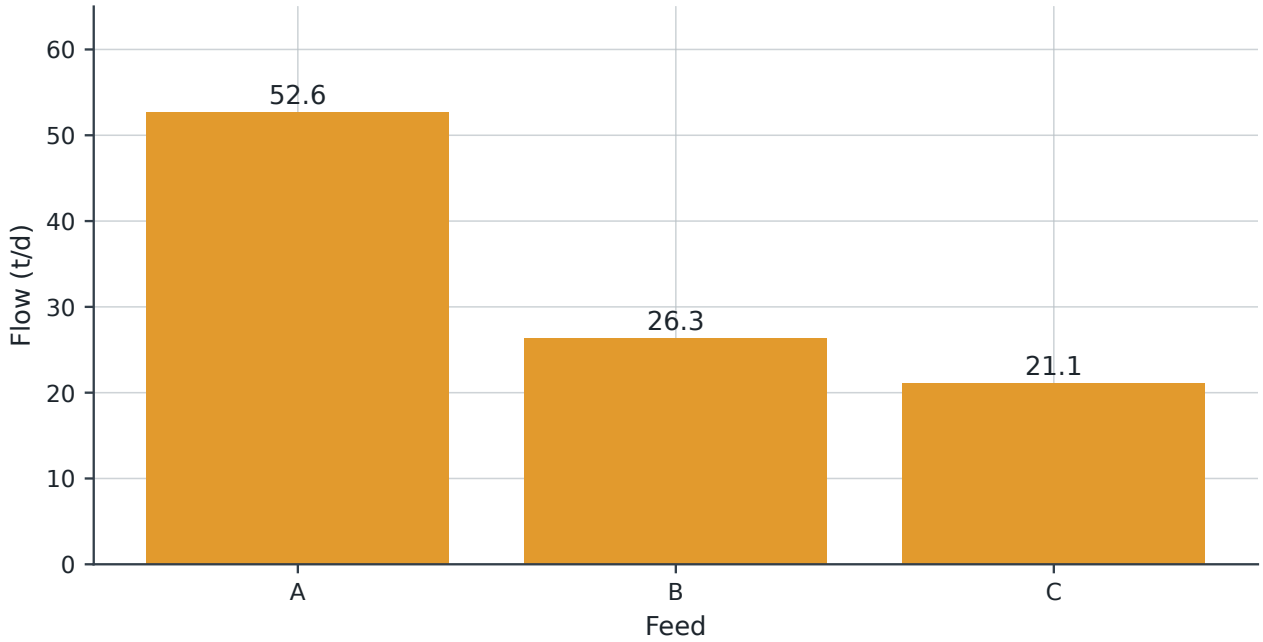

Figure 3.1: Blending with Uncertain Feed Composition: reference solution for level 3.

3.4.1 Check your solution

Check the sorted-term worst case independently of the dual variables. Confirm the two endpoints match Problems 1 and 2.

The automated residual, bound, and integrality audit passed with a maximum violation of 1.11e-16 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

3.4.2 Uncertainty-budget experiment

gamma	cost
0.0000	8125.0000
0.5000	8333.3333
1.0000	8437.5000
1.5000	8526.3158
2.0000	8608.1081
2.5000	8695.6522
3.0000	8771.4286

3.5 Brief Discussion

The protected design costs 8,526.32 USD/d, a premium of 401.32 USD/d (4.94%) over the nominal optimum. Full-box impurity is 7.789%. Protection at a smaller budget must not be described as full-box protection.

3.5.1 Extension to investigate

Propose how historical feed assays could inform the uncertainty set. Discuss correlation and whether the box contains physically implausible combinations.

Part II

Reactor Design and Operation

4 Residence time and throughput

Problem 04 | Reactor Selection and Operating Conditions | Level 1: Guided problem

4.1 Problem Statement

Consider the parallel first-order reactions $A \to D$ and $A \to U$. Desired product D sells for 5 USD/kmol and byproduct U for 1 USD/kmol. Feed costs 1.2 USD/kmol. Fresh feed rate F is at most 100 kmol/h; at least 35 kmol/h of D must be produced. Unreacted feed has no recovery value. Constant density, constant temperature, and equal stoichiometric molar yields are assumed.

The synthetic kinetic functions are

$$k_D = 0.5a \exp[0.025(T - 330)], \qquad k_U = 0.08 \exp[0.055(T - 330)] \quad (\mathrm{h}^{-1}),$$

where T is in K and a is a dimensionless activity factor. These functions are teaching data, not measured kinetics or a full Arrhenius model.

Heating costs $0.006(T - 320)$ USD per kmol of feed. The hourly equipment charge is 20 USD for a PFR or 10 USD for a CSTR. Available feed-equivalent holdup is $F\tau \leq 250$ kmol. If inlet concentration is fixed at 1 kmol/m^3, this corresponds to an equivalent volume limit of 250 m^3. No pressure-drop, heat-transfer, or safety feasibility calculation is included.

Your task. Use a PFR at 330 K with activity $a = 1$. Choose residence time from 1, 2, 3, or 4 h and optimize feed rate and profit.

4.1.1 Before you code

Would maximizing conversion necessarily maximize profit when holdup is limited? Compare the throughput limits at residence times of 2 h and 4 h.

4.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

4.2 Model Formulation

For a mode $j = (r_j, T_j, \tau_j)$, compute $k = k_D + k_U$ and

$$X_j = \begin{cases} 1 - e^{-k\tau_j} & r_j = \text{PFR}, \\ k\tau_j/(1 + k\tau_j) & r_j = \text{CSTR}, \end{cases} \qquad y_{Dj} = X_j k_D/k, \quad y_{Uj} = X_j k_U/k.$$

Let $z_j \in \{0, 1\}$ select a mode and $f_j \geq 0$ be its feed in kmol/h. Require

$$\sum_j z_j = 1, \quad f_j \leq 100 z_j, \quad \sum_j \tau_j f_j \leq 250, \quad \sum_j y_{Dj} f_j \geq 35.$$

The profit for a fixed activity is

$$\Pi = \sum_j \left[5y_{Dj} + y_{Uj} - 1.2 - 0.006(T_j - 320)\right] f_j - \sum_j K_{r_j} z_j.$$

Although the kinetics are nonlinear in temperature and residence time, their values are constants for each listed mode. This gives a MILP that is globally optimal only over the stated finite menu, not over all continuous operating conditions.

Evaluate four modes and maximize Π. Longer residence time raises conversion but reduces the throughput bound $f_j \le 250/\tau_j$. For a fixed mode the profit is linear in feed. Its feasible feed interval is $[35/y_{Dj}, \min(100, 250/\tau_j)]$. An empty interval means the mode cannot meet demand.

4.3 Pyomo Implementation

The code below is the model-building portion of `models/reactors.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.reactors 1
```

```
"""Finite operating-menu reactor design with explicit kinetic assumptions."""

import math
import pyomo.environ as pyo
from models.common import value

FEED_MAX = 100.0  # kmol/h
VOLUME_MAX = 250.0  # flow-residence surrogate in kmol
DEMAND = 35.0  # kmol/h desired product


def yields(kind, temp, tau, activity=1.0):
    kd = activity * 0.5 * math.exp(0.025 * (temp - 330))
    ku = 0.08 * math.exp(0.055 * (temp - 330))
    k = kd + ku
    conversion = 1 - math.exp(-k * tau) if kind == "PFR" else k * tau / (1 + k * tau)
    return conversion * kd / k, conversion * ku / k


def menu(level):
    return [
        (kind, t, tau)
        for kind in (["PFR"] if level == 1 else ["PFR", "CSTR"])
        for t in ([330] if level == 1 else [320, 330, 340, 350])
        for tau in ([1, 2, 3, 4] if level == 1 else [0.5, 1, 2, 3, 4])
    ]


def build(level):
    modes = menu(level)
    states = (
        {"nominal": 1.0} if level < 3 else {"low_activity": 0.8, "high_activity": 1.2}
    )
    m = pyo.ConcreteModel(name="Reactor operating menu")
    m.J = pyo.RangeSet(0, len(modes) - 1)
    m.S = pyo.Set(initialize=list(states))
    m.z = pyo.Var(m.J, domain=pyo.Binary)
    m.f = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.one = pyo.Constraint(expr=sum(m.z[j] for j in m.J) == 1)
    m.bound = pyo.Constraint(m.J, rule=lambda m, j: m.f[j] <= FEED_MAX * m.z[j])
    m.holdup = pyo.Constraint(expr=sum(modes[j][2] * m.f[j] for j in m.J) <= VOLUME_MAX)
    yd = {(s, j): yields(*modes[j], states[s])[0] for s in states for j in m.J}
```

```
    yu = {(s, j): yields(*modes[j], states[s])[1] for s in states for j in m.J}
    m.product = pyo.Constraint(
        m.S, rule=lambda m, s: sum(yd[s, j] * m.f[j] for j in m.J) >= DEMAND
    )

    def profit(m, s):
        return sum(
            (5 * yd[s, j] + yu[s, j] - 1.2 - 0.006 * (modes[j][1] - 320)) * m.f[j]
            - (20 if modes[j][0] == "PFR" else 10) * m.z[j]
            for j in m.J
        )

    m.profit = pyo.Expression(m.S, rule=profit)
    m.eta = pyo.Var(bounds=(-1000, 1000))
    m.floor = pyo.Constraint(m.S, rule=lambda m, s: m.eta <= m.profit[s])
    m.obj = pyo.Objective(expr=m.eta, sense=pyo.maximize)
    m._modes = modes
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(1))
```

4.4 Optimal Solution

The reference objective is **180.6261 USD/h**. This value is optimal for the explicitly stated model and data.

State	Desired (kmol/h)	Undesired (kmol/h)	Profit (USD/h)
nominal	59.2299	9.4768	180.6261

Quantity	Value
Reactor	PFR
Temperature (K)	330.0000
Residence time (h)	3.0000
Feed (kmol/h)	83.3333
Holdup (kmol)	250.0000

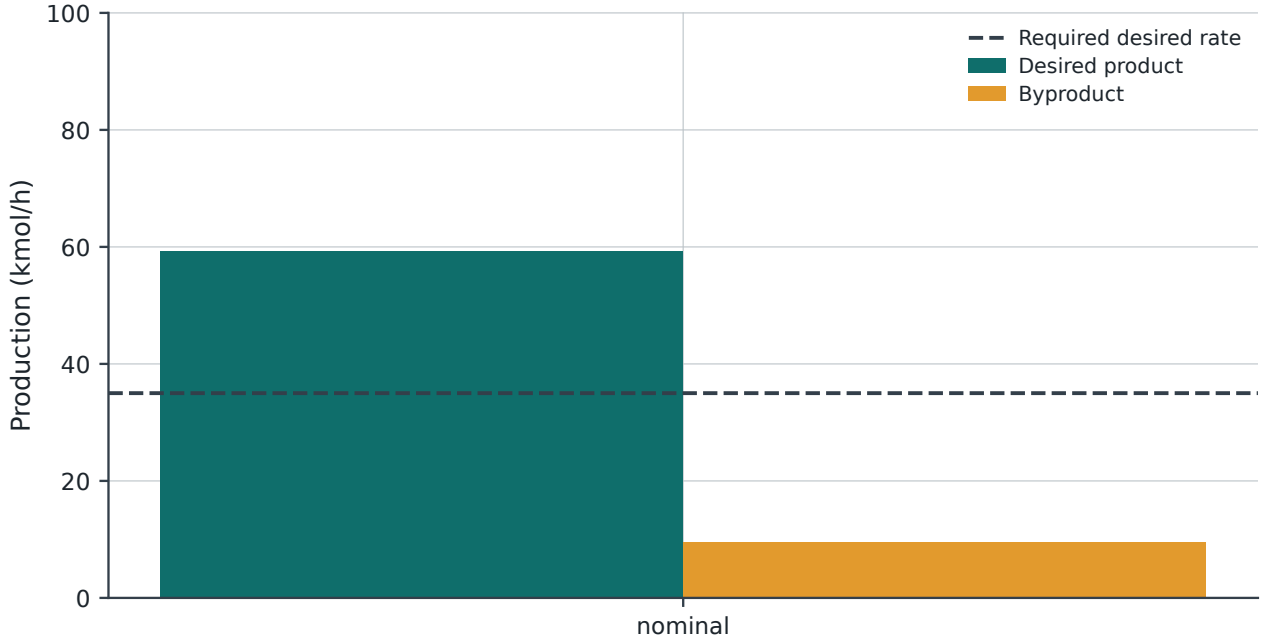

Figure 4.1: Reactor Selection and Operating Conditions: reference solution for level 1.

4.4.1 Check your solution

Enumerate the four modes independently and calculate the optimal endpoint of each feasible feed interval. Confirm $D + U +$ unreacted A $= F$.

The automated residual, bound, and integrality audit passed with a maximum violation of 2.84e-14 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

4.5 Brief Discussion

The selected PFR operates at 330 K and 3.0 h with feed 83.333 kmol/h. The objective is 180.626 USD/h. The feed-holdup tradeoff, product selectivity, and operating cost are included in this choice. Enumerating the same menu independently reproduces the optimum. Refining the menu is a different model, not a stronger claim about this result.

4.5.1 Extension to investigate

Change the desired-product price to 4 USD/kmol. Explain why the preferred residence time may or may not change.

5 Selecting reactor type and temperature

Problem 05 | Reactor Selection and Operating Conditions | Level 2: Model extension

5.1 Problem Statement

Consider the parallel first-order reactions $A \to D$ and $A \to U$. Desired product D sells for 5 USD/kmol and byproduct U for 1 USD/kmol. Feed costs 1.2 USD/kmol. Fresh feed rate F is at most 100 kmol/h; at least 35 kmol/h of D must be produced. Unreacted feed has no recovery value. Constant density, constant temperature, and equal stoichiometric molar yields are assumed.

The synthetic kinetic functions are

$$k_D = 0.5a \exp[0.025(T - 330)], \qquad k_U = 0.08 \exp[0.055(T - 330)] \quad (\mathrm{h}^{-1}),$$

where T is in K and a is a dimensionless activity factor. These functions are teaching data, not measured kinetics or a full Arrhenius model.

Heating costs $0.006(T - 320)$ USD per kmol of feed. The hourly equipment charge is 20 USD for a PFR or 10 USD for a CSTR. Available feed-equivalent holdup is $F\tau \leq 250$ kmol. If inlet concentration is fixed at 1 kmol/m^3, this corresponds to an equivalent volume limit of 250 m^3. No pressure-drop, heat-transfer, or safety feasibility calculation is included.

Your task. Allow both PFR and CSTR designs. Use temperatures 320, 330, 340, and 350 K and residence times 0.5, 1, 2, 3, and 4 h. Activity remains 1. Optimize the 40-mode design.

5.1.1 Before you code

The undesired reaction accelerates more sharply with temperature. Does a higher temperature always improve the desired-product yield? Can a cheaper CSTR offset lower conversion?

5.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

5.2 Model Formulation

For a mode $j = (r_j, T_j, \tau_j)$, compute $k = k_D + k_U$ and

$$X_j = \begin{cases} 1 - e^{-k\tau_j} & r_j = \text{PFR}, \\ k\tau_j/(1 + k\tau_j) & r_j = \text{CSTR}, \end{cases} \qquad y_{Dj} = X_j k_D/k, \quad y_{Uj} = X_j k_U/k.$$

Let $z_j \in \{0, 1\}$ select a mode and $f_j \geq 0$ be its feed in kmol/h. Require

$$\sum_j z_j = 1, \quad f_j \leq 100 z_j, \quad \sum_j \tau_j f_j \leq 250, \quad \sum_j y_{Dj} f_j \geq 35.$$

The profit for a fixed activity is

$$\Pi = \sum_j \left[5y_{Dj} + y_{Uj} - 1.2 - 0.006(T_j - 320)\right] f_j - \sum_j K_{r_j} z_j.$$

Although the kinetics are nonlinear in temperature and residence time, their values are constants for each listed mode. This gives a MILP that is globally optimal only over the stated finite menu, not over all continuous operating conditions.

Use the same MILP with 40 candidate modes and their corresponding equipment charges. The binary choice makes reactor selection mutually exclusive. Feed is disaggregated by mode, so no product of a binary variable and a continuous feed variable is needed in the objective or holdup constraint.

5.3 Pyomo Implementation

The code below is the model-building portion of `models/reactors.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.reactors 2
```

```
"""Finite operating-menu reactor design with explicit kinetic assumptions."""

import math
import pyomo.environ as pyo
from models.common import value

FEED_MAX = 100.0  # kmol/h
VOLUME_MAX = 250.0  # flow-residence surrogate in kmol
DEMAND = 35.0  # kmol/h desired product


def yields(kind, temp, tau, activity=1.0):
    kd = activity * 0.5 * math.exp(0.025 * (temp - 330))
    ku = 0.08 * math.exp(0.055 * (temp - 330))
    k = kd + ku
    conversion = 1 - math.exp(-k * tau) if kind == "PFR" else k * tau / (1 + k * tau)
    return conversion * kd / k, conversion * ku / k


def menu(level):
    return [
        (kind, t, tau)
        for kind in (["PFR"] if level == 1 else ["PFR", "CSTR"])
        for t in ([330] if level == 1 else [320, 330, 340, 350])
        for tau in ([1, 2, 3, 4] if level == 1 else [0.5, 1, 2, 3, 4])
    ]


def build(level):
    modes = menu(level)
    states = (
        {"nominal": 1.0} if level < 3 else {"low_activity": 0.8, "high_activity": 1.2}
    )
    m = pyo.ConcreteModel(name="Reactor operating menu")
    m.J = pyo.RangeSet(0, len(modes) - 1)
    m.S = pyo.Set(initialize=list(states))
    m.z = pyo.Var(m.J, domain=pyo.Binary)
    m.f = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.one = pyo.Constraint(expr=sum(m.z[j] for j in m.J) == 1)
    m.bound = pyo.Constraint(m.J, rule=lambda m, j: m.f[j] <= FEED_MAX * m.z[j])
    m.holdup = pyo.Constraint(expr=sum(modes[j][2] * m.f[j] for j in m.J) <= VOLUME_MAX)
    yd = {(s, j): yields(*modes[j], states[s])[0] for s in states for j in m.J}
```

```
    yu = {(s, j): yields(*modes[j], states[s])[1] for s in states for j in m.J}
    m.product = pyo.Constraint(
        m.S, rule=lambda m, s: sum(yd[s, j] * m.f[j] for j in m.J) >= DEMAND
    )

    def profit(m, s):
        return sum(
            (5 * yd[s, j] + yu[s, j] - 1.2 - 0.006 * (modes[j][1] - 320)) * m.f[j]
            - (20 if modes[j][0] == "PFR" else 10) * m.z[j]
            for j in m.J
        )

    m.profit = pyo.Expression(m.S, rule=profit)
    m.eta = pyo.Var(bounds=(-1000, 1000))
    m.floor = pyo.Constraint(m.S, rule=lambda m, s: m.eta <= m.profit[s])
    m.obj = pyo.Objective(expr=m.eta, sense=pyo.maximize)
    m._modes = modes
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(2))
```

5.4 Optimal Solution

The reference objective is **202.9896 USD/h**. This value is optimal for the explicitly stated model and data.

State	Desired (kmol/h)	Undesired (kmol/h)	Profit (USD/h)
nominal	68.2202	19.8888	202.9896

Quantity	Value
Reactor	PFR
Temperature (K)	350.0000
Residence time (h)	2.0000
Feed (kmol/h)	100.0000
Holdup (kmol)	200.0000

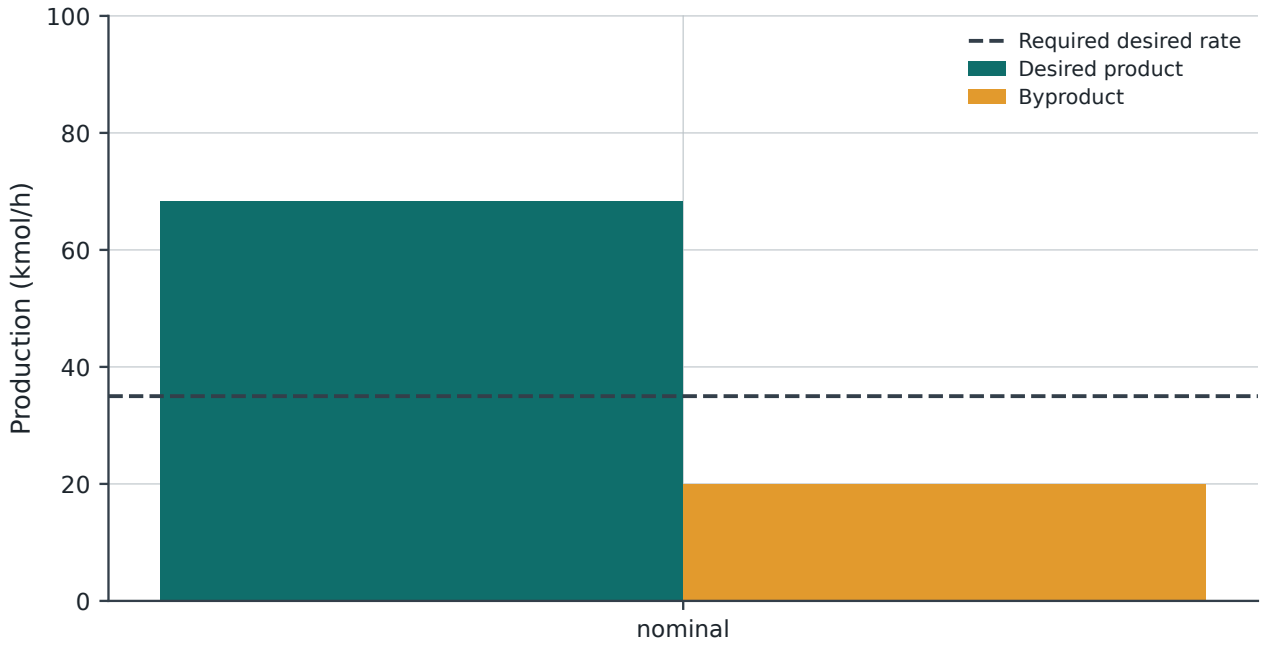

Figure 5.1: Reactor Selection and Operating Conditions: reference solution for level 2.

5.4.1 Check your solution

Check the winning mode against independent enumeration of all 40 one-dimensional feed problems. State the grid limitation in the conclusion.

The automated residual, bound, and integrality audit passed with a maximum violation of 5.68e-14 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

5.5 Brief Discussion

The selected PFR operates at 350 K and 2.0 h with feed 100.000 kmol/h. The objective is 202.990 USD/h. The feed-holdup tradeoff, product selectivity, and operating cost are included in this choice. Enumerating the same menu independently reproduces the optimum. Refining the menu is a different model, not a stronger claim about this result.

5.5.1 Extension to investigate

Refine the temperature and residence-time grids near the selected mode. Quantify the profit change, and distinguish grid convergence from a proof of the continuous global optimum.

6 A design robust to catalyst activity

Problem 06 | Reactor Selection and Operating Conditions | Level 3: Open-ended challenge

6.1 Problem Statement

Consider the parallel first-order reactions $A \to D$ and $A \to U$. Desired product D sells for 5 USD/kmol and byproduct U for 1 USD/kmol. Feed costs 1.2 USD/kmol. Fresh feed rate F is at most 100 kmol/h; at least 35 kmol/h of D must be produced. Unreacted feed has no recovery value. Constant density, constant temperature, and equal stoichiometric molar yields are assumed.

The synthetic kinetic functions are

$$k_D = 0.5a \exp[0.025(T - 330)], \qquad k_U = 0.08 \exp[0.055(T - 330)] \quad (\mathrm{h}^{-1}),$$

where T is in K and a is a dimensionless activity factor. These functions are teaching data, not measured kinetics or a full Arrhenius model.

Heating costs $0.006(T - 320)$ USD per kmol of feed. The hourly equipment charge is 20 USD for a PFR or 10 USD for a CSTR. Available feed-equivalent holdup is $F\tau \leq 250$ kmol. If inlet concentration is fixed at 1 kmol/m^3, this corresponds to an equivalent volume limit of 250 m^3. No pressure-drop, heat-transfer, or safety feasibility calculation is included.

Your task. Keep the 40-mode menu. Activity is either 0.8 or 1.2. Choose one common reactor, temperature, residence time, and feed rate before activity is known. Meet the product requirement in both cases and maximize the smaller hourly profit. Explain whether fixed operation or adjustable operation is more appropriate for a real plant.

6.1.1 Before you code

If the plant can measure activity before selecting feed rate, which decision would become scenario dependent? Would that provide a different guarantee?

6.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

6.2 Model Formulation

For a mode $j = (r_j, T_j, \tau_j)$, compute $k = k_D + k_U$ and

$$X_j = \begin{cases} 1 - e^{-k\tau_j} & r_j = \text{PFR}, \\ k\tau_j/(1 + k\tau_j) & r_j = \text{CSTR}, \end{cases} \qquad y_{Dj} = X_j k_D/k, \quad y_{Uj} = X_j k_U/k.$$

Let $z_j \in \{0, 1\}$ select a mode and $f_j \geq 0$ be its feed in kmol/h. Require

$$\sum_j z_j = 1, \quad f_j \leq 100 z_j, \quad \sum_j \tau_j f_j \leq 250, \quad \sum_j y_{Dj} f_j \geq 35.$$

The profit for a fixed activity is

$$\Pi = \sum_j \left[5y_{Dj} + y_{Uj} - 1.2 - 0.006(T_j - 320)\right] f_j - \sum_j K_{r_j} z_j.$$

Although the kinetics are nonlinear in temperature and residence time, their values are constants for each listed mode. This gives a MILP that is globally optimal only over the stated finite menu, not over all continuous operating conditions.

For each state s, calculate y_{Djs} and y_{Ujs}. Introduce a guaranteed-profit variable η and solve

$$\max \eta, \qquad \eta \le \Pi_s \quad \forall s, \qquad \sum_j y_{Djs} f_j \ge 35 \quad \forall s.$$

All z_j and f_j are shared across states, giving a static robust design. There are no scenario probabilities. Independent enumeration remains possible because the equipment charge for a mode is the same in both states and the lower profit slope determines its worst-case profit.

6.3 Pyomo Implementation

The code below is the model-building portion of `models/reactors.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.reactors 3
```

```
"""Finite operating-menu reactor design with explicit kinetic assumptions."""

import math
import pyomo.environ as pyo
from models.common import value

FEED_MAX = 100.0  # kmol/h
VOLUME_MAX = 250.0  # flow-residence surrogate in kmol
DEMAND = 35.0  # kmol/h desired product


def yields(kind, temp, tau, activity=1.0):
    kd = activity * 0.5 * math.exp(0.025 * (temp - 330))
    ku = 0.08 * math.exp(0.055 * (temp - 330))
    k = kd + ku
    conversion = 1 - math.exp(-k * tau) if kind == "PFR" else k * tau / (1 + k * tau)
    return conversion * kd / k, conversion * ku / k


def menu(level):
    return [
        (kind, t, tau)
        for kind in (["PFR"] if level == 1 else ["PFR", "CSTR"])
        for t in ([330] if level == 1 else [320, 330, 340, 350])
        for tau in ([1, 2, 3, 4] if level == 1 else [0.5, 1, 2, 3, 4])
    ]


def build(level):
    modes = menu(level)
    states = (
        {"nominal": 1.0} if level < 3 else {"low_activity": 0.8, "high_activity": 1.2}
    )
    m = pyo.ConcreteModel(name="Reactor operating menu")
    m.J = pyo.RangeSet(0, len(modes) - 1)
    m.S = pyo.Set(initialize=list(states))
    m.z = pyo.Var(m.J, domain=pyo.Binary)
```

```
    m.f = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.one = pyo.Constraint(expr=sum(m.z[j] for j in m.J) == 1)
    m.bound = pyo.Constraint(m.J, rule=lambda m, j: m.f[j] <= FEED_MAX * m.z[j])
    m.holdup = pyo.Constraint(expr=sum(modes[j][2] * m.f[j] for j in m.J) <= VOLUME_MAX)
    yd = {(s, j): yields(*modes[j], states[s])[0] for s in states for j in m.J}
    yu = {(s, j): yields(*modes[j], states[s])[1] for s in states for j in m.J}
    m.product = pyo.Constraint(
        m.S, rule=lambda m, s: sum(yd[s, j] * m.f[j] for j in m.J) >= DEMAND
    )

    def profit(m, s):
        return sum(
            (5 * yd[s, j] + yu[s, j] - 1.2 - 0.006 * (modes[j][1] - 320)) * m.f[j]
            - (20 if modes[j][0] == "PFR" else 10) * m.z[j]
            for j in m.J
        )

    m.profit = pyo.Expression(m.S, rule=profit)
    m.eta = pyo.Var(bounds=(-1000, 1000))
    m.floor = pyo.Constraint(m.S, rule=lambda m, s: m.eta <= m.profit[s])
    m.obj = pyo.Objective(expr=m.eta, sense=pyo.maximize)
    m._modes = modes
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(3))
```

6.4 Optimal Solution

The reference objective is **170.6059 USD/h**. This value is optimal for the explicitly stated model and data.

State	Desired (kmol/h)	Undesired (kmol/h)	Profit (USD/h)
low_activity	56.9690	20.7609	170.6059
high_activity	65.3684	15.8812	207.7233

Quantity	Value
Reactor	PFR
Temperature (K)	350.0000
Residence time (h)	3.0000
Feed (kmol/h)	83.3333
Holdup (kmol)	250.0000

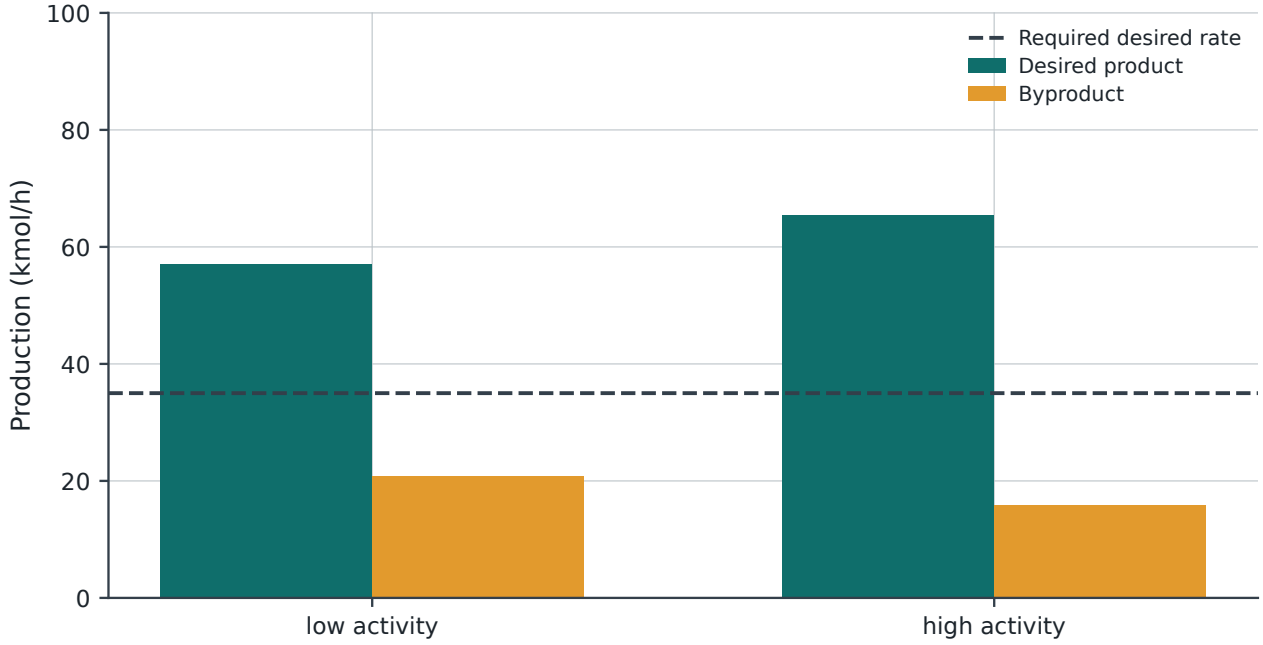

Figure 6.1: Reactor Selection and Operating Conditions: reference solution for level 3.

6.4.1 Check your solution

Check mass balance and minimum desired-product rate in both states. Compare the nominal optimum evaluated under low activity with the robust optimum evaluated under that same activity.

The automated residual, bound, and integrality audit passed with a maximum violation of 2.84e-14 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

6.5 Brief Discussion

The selected PFR operates at 350 K and 3.0 h with feed 83.333 kmol/h. The objective is 170.606 USD/h. This is the guaranteed profit across the two activity states; state-specific profit can be higher. Enumerating the same menu independently reproduces the optimum. Refining the menu is a different model, not a stronger claim about this result.

6.5.1 Extension to investigate

Formulate adjustable feed rates f_{js} with a shared mode z_j. Explain how the information timing changes the feasible set and whether the worst-case value can decrease.

Part III

Heat Recovery

7 Dispatch through existing exchangers

Problem 07 | Heat Recovery Network Design | Level 1: Guided problem

7.1 Problem Statement

Two fixed-temperature heat sources provide 120 kW at 180 °C (H1) and 80 kW at 110 °C (H2). Heat users require 100 kW at 150 °C (C1) and 100 kW at 80 °C (C2). A minimum approach of 20 °C allows H1-C1, H1-C2, and H2-C2; H2-C1 is forbidden. These are isothermal source/sink surrogates, not sensible-heat streams with temperature profiles.

Match	Maximum recovered heat (kW)	Daily installation charge (USD/d)
H1-C1	100	25
H1-C2	90	20
H2-C2	70	18

Hot utility costs 1 USD/(kW d); cold utility costs 0.15 USD/(kW d). These coefficients already convert a constant daily heat rate to a daily cost. Do not multiply by 24 again. Utilities are available at suitable temperatures without capacity limits. All unused source heat must be rejected.

Your task. Assume all three exchangers are already installed and their installation charges are sunk. Minimize daily utility cost.

7.1.1 Before you code

Total source heat equals total demand. Does that guarantee zero hot utility when match capacities and temperature feasibility are respected?

7.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

7.2 Model Formulation

Let $q_{hcs} \geq 0$ be recovered heat on an allowed match, $u_{cs} \geq 0$ hot utility, and $v_{hs} \geq 0$ rejected heat, all in kW. Let $z_{hc} \in \{0, 1\}$ select an exchanger. For each operating state s,

$$\sum_{c} q_{hcs} + v_{hs} = H_h, \qquad \sum_{h} q_{hcs} + u_{cs} = D_{cs}, \qquad q_{hcs} \leq \overline{Q}_{hc} z_{hc}.$$

Only allowed matches appear in the sums. The first balance accounts for surplus heat, while the second prevents counting recovered heat twice. Installation charges are daily equivalents of ownership costs, not one-time capital expenses.

Fix every $z_{hc} = 1$ and solve the LP

$$\min C = \sum_{c} u_c + 0.15 \sum_{h} v_h.$$

The network has 200 kW on each side, but the H2-C2 limit prevents full recovery. Simultaneous hot and cold utility is therefore possible even with equal aggregate supply and demand.

7.3 Pyomo Implementation

The code below is the model-building portion of `models/heat.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.heat 1
```

```
"""Heat recovery between fixed-temperature utility-like streams."""

import pyomo.environ as pyo
from models.common import value

HOT = {"H1": 120, "H2": 80}  # kW available
ARCS = [("H1", "C1"), ("H1", "C2"), ("H2", "C2")]
CAP = {("H1", "C1"): 100, ("H1", "C2"): 90, ("H2", "C2"): 70}
FIXED = {("H1", "C1"): 25, ("H1", "C2"): 20, ("H2", "C2"): 18}  # USD/d


def build(level):
    states = (
        {"normal": (100, 100, 1.0, 1.0)}
        if level < 3
        else {"low": (100, 80, 1.0, 0.5), "high": (100, 130, 2.0, 0.5)}
    )
    m = pyo.ConcreteModel(name="Fixed-temperature heat recovery")
    m.A = pyo.Set(initialize=ARCS, dimen=2)
    m.H = pyo.Set(initialize=list(HOT))
    m.C = pyo.Set(initialize=["C1", "C2"])
    m.S = pyo.Set(initialize=list(states))
    m.z = pyo.Var(m.A, domain=pyo.Binary)
    if level == 1:
        for a in m.A:
            m.z[a].fix(1)
    m.q = pyo.Var(m.S, m.A, domain=pyo.NonNegativeReals)
    m.hu = pyo.Var(m.S, m.C, domain=pyo.NonNegativeReals)
    m.cu = pyo.Var(m.S, m.H, domain=pyo.NonNegativeReals)
    m.hot = pyo.Constraint(
        m.S,
        m.H,
        rule=lambda m, s, h: (
            sum(m.q[s, i, j] for i, j in m.A if i == h) + m.cu[s, h] == HOT[h]
        ),
    )
    m.cold = pyo.Constraint(
        m.S,
        m.C,
        rule=lambda m, s, c: (
            sum(m.q[s, i, j] for i, j in m.A if j == c) + m.hu[s, c]
            == states[s][0 if c == "C1" else 1]
        ),
    )
    m.cap = pyo.Constraint(
        m.S, m.A, rule=lambda m, s, h, c: m.q[s, h, c] <= CAP[h, c] * m.z[h, c]
    )
```

```
    if level == 3:
        m.budget = pyo.Constraint(expr=sum(FIXED[a] * m.z[a] for a in m.A) <= 45)
    fixed = 0 if level == 1 else sum(FIXED[a] * m.z[a] for a in m.A)
    m.obj = pyo.Objective(
        expr=fixed
        + sum(
            states[s][3]
            * (
                states[s][2] * sum(m.hu[s, c] for c in m.C)
                + 0.15 * sum(m.cu[s, h] for h in m.H)
            )
            for s in m.S
        )
    )
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(1))
```

7.4 Optimal Solution

The reference objective is **11.5000 USD/d**. This value is optimal for the explicitly stated model and data.

State	Match	Installed	Heat (kW)
normal	H1 to C1	1	100.0000
normal	H1 to C2	1	20.0000
normal	H2 to C2	1	70.0000

Quantity	Value
normal: hot utility (kW)	10.0000
normal: cold utility (kW)	10.0000

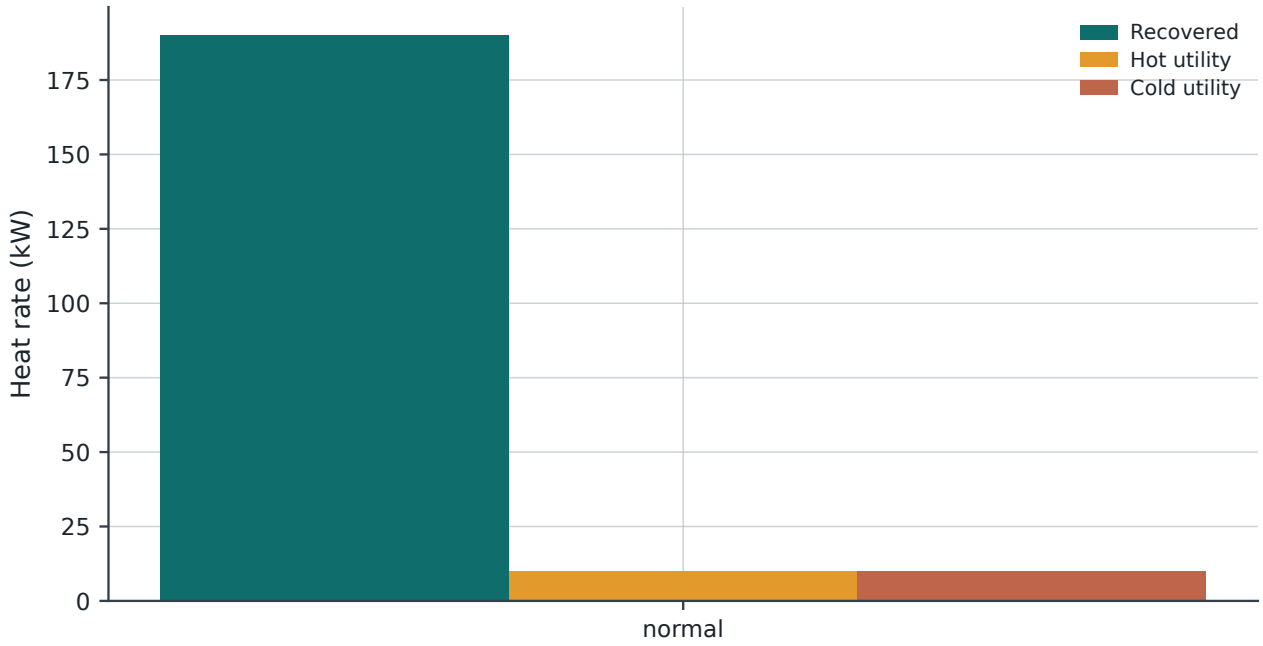

Figure 7.1: Heat Recovery Network Design: reference solution for level 1.

7.4.1 Check your solution

Calculate recovered heat and both utility totals. Confirm hot utility − cold utility = total demand − total source heat.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

7.5 Brief Discussion

The objective is 11.50 USD/d. The existing system still needs 10 kW of hot utility and rejects 10 kW because the match limits prevent full recovery. For a physical exchanger network with varying stream temperatures, stagewise temperature balances and minimum approaches would need to replace this fixed-temperature representation.

7.5.1 Extension to investigate

Increase the H2-C2 capacity from 70 to 80 kW. Predict the change in both utilities and daily cost before solving.

8 Selecting which exchangers to install

Problem 08 | Heat Recovery Network Design | Level 2: Model extension

8.1 Problem Statement

Two fixed-temperature heat sources provide 120 kW at 180 °C (H1) and 80 kW at 110 °C (H2). Heat users require 100 kW at 150 °C (C1) and 100 kW at 80 °C (C2). A minimum approach of 20 °C allows H1-C1, H1-C2, and H2-C2; H2-C1 is forbidden. These are isothermal source/sink surrogates, not sensible-heat streams with temperature profiles.

Match	Maximum recovered heat (kW)	Daily installation charge (USD/d)
H1-C1	100	25
H1-C2	90	20
H2-C2	70	18

Hot utility costs 1 USD/(kW d); cold utility costs 0.15 USD/(kW d). These coefficients already convert a constant daily heat rate to a daily cost. Do not multiply by 24 again. Utilities are available at suitable temperatures without capacity limits. All unused source heat must be rejected.

Your task. Now treat every exchanger as a new investment. Minimize installation charges plus utility cost. Compare the chosen design with the existing-network result without interpreting sunk-cost and investment objectives as identical.

8.1.1 Before you code

Can an exchanger with a small heat duty still pay for its fixed charge? What must be compared to judge its value?

8.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

8.2 Model Formulation

Let $q_{hcs} \geq 0$ be recovered heat on an allowed match, $u_{cs} \geq 0$ hot utility, and $v_{hs} \geq 0$ rejected heat, all in kW. Let $z_{hc} \in \{0, 1\}$ select an exchanger. For each operating state s,

$$\sum_{c} q_{hcs} + v_{hs} = H_h, \qquad \sum_{h} q_{hcs} + u_{cs} = D_{cs}, \qquad q_{hcs} \leq \overline{Q}_{hc} z_{hc}.$$

Only allowed matches appear in the sums. The first balance accounts for surplus heat, while the second prevents counting recovered heat twice. Installation charges are daily equivalents of ownership costs, not one-time capital expenses.

Release the binaries and minimize

$$C = \sum_{(h,c)} K_{hc} z_{hc} + \sum_{c} u_c + 0.15 \sum_{h} v_h.$$

The capacity implication is exact because the exchanger maximum is a physical upper bound. No arbitrary large constant is needed. A zero-duty exchanger may be excluded to avoid a positive fixed charge.

8.3 Pyomo Implementation

The code below is the model-building portion of `models/heat.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.heat 2
```

```
"""Heat recovery between fixed-temperature utility-like streams."""

import pyomo.environ as pyo
from models.common import value

HOT = {"H1": 120, "H2": 80}  # kW available
ARCS = [("H1", "C1"), ("H1", "C2"), ("H2", "C2")]
CAP = {("H1", "C1"): 100, ("H1", "C2"): 90, ("H2", "C2"): 70}
FIXED = {("H1", "C1"): 25, ("H1", "C2"): 20, ("H2", "C2"): 18}  # USD/d


def build(level):
    states = (
        {"normal": (100, 100, 1.0, 1.0)}
        if level < 3
        else {"low": (100, 80, 1.0, 0.5), "high": (100, 130, 2.0, 0.5)}
    )
    m = pyo.ConcreteModel(name="Fixed-temperature heat recovery")
    m.A = pyo.Set(initialize=ARCS, dimen=2)
    m.H = pyo.Set(initialize=list(HOT))
    m.C = pyo.Set(initialize=["C1", "C2"])
    m.S = pyo.Set(initialize=list(states))
    m.z = pyo.Var(m.A, domain=pyo.Binary)
    if level == 1:
        for a in m.A:
            m.z[a].fix(1)
    m.q = pyo.Var(m.S, m.A, domain=pyo.NonNegativeReals)
    m.hu = pyo.Var(m.S, m.C, domain=pyo.NonNegativeReals)
    m.cu = pyo.Var(m.S, m.H, domain=pyo.NonNegativeReals)
    m.hot = pyo.Constraint(
        m.S,
        m.H,
        rule=lambda m, s, h: (
            sum(m.q[s, i, j] for i, j in m.A if i == h) + m.cu[s, h] == HOT[h]
        ),
    )
    m.cold = pyo.Constraint(
        m.S,
        m.C,
        rule=lambda m, s, c: (
            sum(m.q[s, i, j] for i, j in m.A if j == c) + m.hu[s, c]
            == states[s][0 if c == "C1" else 1]
        ),
    )
    m.cap = pyo.Constraint(
        m.S, m.A, rule=lambda m, s, h, c: m.q[s, h, c] <= CAP[h, c] * m.z[h, c]
    )
```

```
    if level == 3:
        m.budget = pyo.Constraint(expr=sum(FIXED[a] * m.z[a] for a in m.A) <= 45)
    fixed = 0 if level == 1 else sum(FIXED[a] * m.z[a] for a in m.A)
    m.obj = pyo.Objective(
        expr=fixed
        + sum(
            states[s][3]
            * (
                states[s][2] * sum(m.hu[s, c] for c in m.C)
                + 0.15 * sum(m.cu[s, h] for h in m.H)
            )
            for s in m.S
        )
    )
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(2))
```

8.4 Optimal Solution

The reference objective is **74.5000 USD/d**. This value is optimal for the explicitly stated model and data.

State	Match	Installed	Heat (kW)
normal	H1 to C1	1	90.0000
normal	H1 to C2	1	30.0000
normal	H2 to C2	1	70.0000

Quantity	Value
normal: hot utility (kW)	10.0000
normal: cold utility (kW)	10.0000

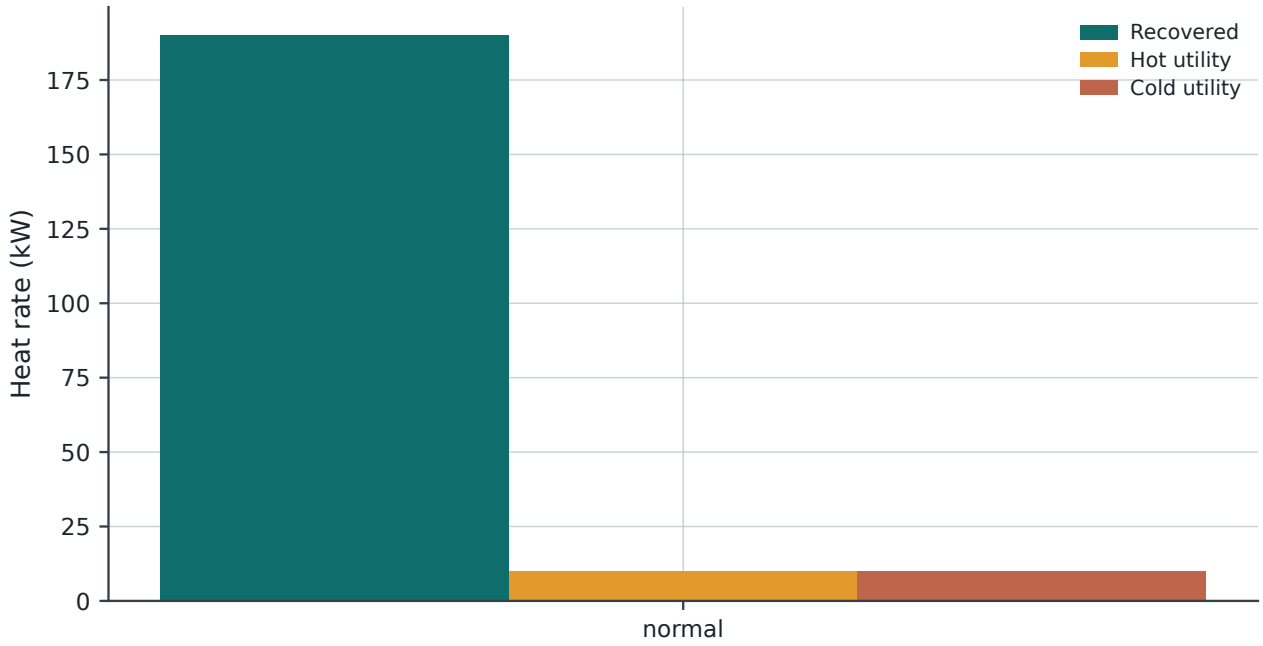

Figure 8.1: Heat Recovery Network Design: reference solution for level 2.

8.4.1 Check your solution

Enumerate the eight installation patterns and solve the residual heat-allocation LP for each. Compare the best value with the MILP.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

8.5 Brief Discussion

The objective is 74.50 USD/d. The objective includes ownership charges, so its increase relative to the existing-system LP is not an efficiency loss. For a physical exchanger network with varying stream temperatures, stagewise temperature balances and minimum approaches would need to replace this fixed-temperature representation.

8.5.1 Extension to investigate

Raise the H1-C2 daily charge. Find the threshold at which that exchanger is no longer economic and verify the two neighboring designs.

9 A shared design for two operating states

Problem 09 | Heat Recovery Network Design | Level 3: Open-ended challenge

9.1 Problem Statement

Two fixed-temperature heat sources provide 120 kW at 180 °C (H1) and 80 kW at 110 °C (H2). Heat users require 100 kW at 150 °C (C1) and 100 kW at 80 °C (C2). A minimum approach of 20 °C allows H1-C1, H1-C2, and H2-C2; H2-C1 is forbidden. These are isothermal source/sink surrogates, not sensible-heat streams with temperature profiles.

Match	Maximum recovered heat (kW)	Daily installation charge (USD/d)
H1-C1	100	25
H1-C2	90	20
H2-C2	70	18

Hot utility costs 1 USD/(kW d); cold utility costs 0.15 USD/(kW d). These coefficients already convert a constant daily heat rate to a daily cost. Do not multiply by 24 again. Utilities are available at suitable temperatures without capacity limits. All unused source heat must be rejected.

Your task. Choose exchangers before the operating state is known. In the low state, C2 requires 80 kW and hot utility costs 1 USD/(kW d). In the high state, C2 requires 130 kW and hot utility costs 2 USD/(kW d). Both states have probability 0.5. C1 remains at 100 kW. Limit total daily installation charges to 45 USD/d. Operating flows may adapt after the state is observed.

9.1.1 Before you code

Which variables must be shared across states? Why is solving each state independently and averaging the costs too optimistic for one physical network?

9.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

9.2 Model Formulation

Let $q_{hcs} \geq 0$ be recovered heat on an allowed match, $u_{cs} \geq 0$ hot utility, and $v_{hs} \geq 0$ rejected heat, all in kW. Let $z_{hc} \in \{0, 1\}$ select an exchanger. For each operating state s,

$$\sum_{c} q_{hcs} + v_{hs} = H_h, \qquad \sum_{h} q_{hcs} + u_{cs} = D_{cs}, \qquad q_{hcs} \leq \overline{Q}_{hc} z_{hc}.$$

Only allowed matches appear in the sums. The first balance accounts for surplus heat, while the second prevents counting recovered heat twice. Installation charges are daily equivalents of ownership costs, not one-time capital expenses.

Use shared z_{hc}, state-specific heat and utilities, and

$$\sum_{h,c} K_{hc} z_{hc} \le 45,$$

$$\min C = \sum_{h,c} K_{hc} z_{hc} + \sum_{s} \pi_s \left(c_s^H \sum_{c} u_{cs} + 0.15 \sum_{h} v_{hs} \right).$$

The fixed charge appears once. The objective includes expected operating cost, while the two sets of heat balances ensure feasibility in both states. This is a two-stage stochastic MILP for a finite set of states.

9.3 Pyomo Implementation

The code below is the model-building portion of `models/heat.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.heat 3
```

```
"""Heat recovery between fixed-temperature utility-like streams."""

import pyomo.environ as pyo
from models.common import value

HOT = {"H1": 120, "H2": 80}  # kW available
ARCS = [("H1", "C1"), ("H1", "C2"), ("H2", "C2")]
CAP = {("H1", "C1"): 100, ("H1", "C2"): 90, ("H2", "C2"): 70}
FIXED = {("H1", "C1"): 25, ("H1", "C2"): 20, ("H2", "C2"): 18}  # USD/d


def build(level):
    states = (
        {"normal": (100, 100, 1.0, 1.0)}
        if level < 3
        else {"low": (100, 80, 1.0, 0.5), "high": (100, 130, 2.0, 0.5)}
    )
    m = pyo.ConcreteModel(name="Fixed-temperature heat recovery")
    m.A = pyo.Set(initialize=ARCS, dimen=2)
    m.H = pyo.Set(initialize=list(HOT))
    m.C = pyo.Set(initialize=["C1", "C2"])
    m.S = pyo.Set(initialize=list(states))
    m.z = pyo.Var(m.A, domain=pyo.Binary)
    if level == 1:
        for a in m.A:
            m.z[a].fix(1)
    m.q = pyo.Var(m.S, m.A, domain=pyo.NonNegativeReals)
    m.hu = pyo.Var(m.S, m.C, domain=pyo.NonNegativeReals)
    m.cu = pyo.Var(m.S, m.H, domain=pyo.NonNegativeReals)
    m.hot = pyo.Constraint(
        m.S,
        m.H,
        rule=lambda m, s, h: (
            sum(m.q[s, i, j] for i, j in m.A if i == h) + m.cu[s, h] == HOT[h]
        ),
    )
    m.cold = pyo.Constraint(
        m.S,
        m.C,
        rule=lambda m, s, c: (
            sum(m.q[s, i, j] for i, j in m.A if j == c) + m.hu[s, c]
            == states[s][0 if c == "C1" else 1]
        ),
```

```
    )
    m.cap = pyo.Constraint(
        m.S, m.A, rule=lambda m, s, h, c: m.q[s, h, c] <= CAP[h, c] * m.z[h, c]
    )
    if level == 3:
        m.budget = pyo.Constraint(expr=sum(FIXED[a] * m.z[a] for a in m.A) <= 45)
    fixed = 0 if level == 1 else sum(FIXED[a] * m.z[a] for a in m.A)
    m.obj = pyo.Objective(
        expr=fixed
        + sum(
            states[s][3]
            * (
                states[s][2] * sum(m.hu[s, c] for c in m.C)
                + 0.15 * sum(m.cu[s, h] for h in m.H)
            )
            for s in m.S
        )
    )
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(3))
```

9.4 Optimal Solution

The reference objective is **112.5000 USD/d**. This value is optimal for the explicitly stated model and data.

State	Match	Installed	Heat (kW)
low	H1 to C1	1	100.0000
low	H1 to C2	0	0.0000
low	H2 to C2	1	70.0000
high	H1 to C1	1	100.0000
high	H1 to C2	0	0.0000
high	H2 to C2	1	70.0000

Quantity	Value
low: hot utility (kW)	10.0000
high: hot utility (kW)	60.0000
low: cold utility (kW)	30.0000
high: cold utility (kW)	30.0000

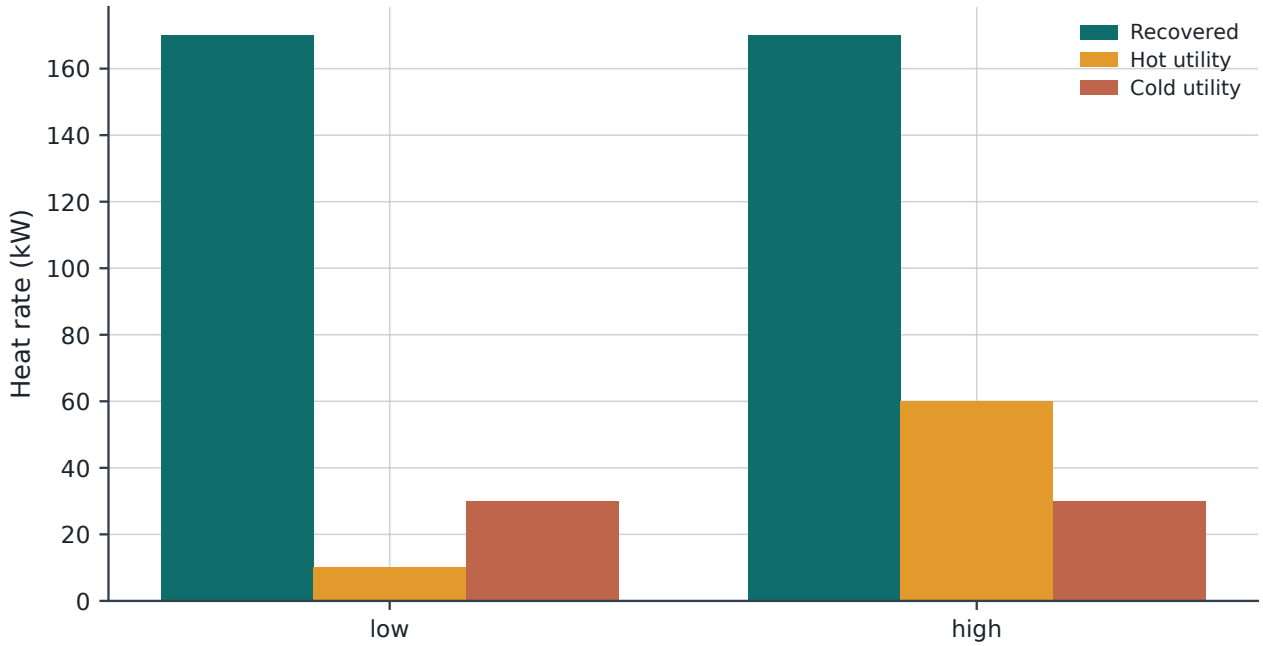

Figure 9.1: Heat Recovery Network Design: reference solution for level 3.

9.4.1 Check your solution

Check energy balances separately in both states and verify the shared design fits the budget. Enumerate budget-feasible installation patterns.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

9.5 Brief Discussion

The objective is 112.50 USD/d. The objective includes ownership charges, so its increase relative to the existing-system LP is not an efficiency loss. For a physical exchanger network with varying stream temperatures, stagewise temperature balances and minimum approaches would need to replace this fixed-temperature representation.

9.5.1 Extension to investigate

Recommend a design criterion if the high state becomes rare but disruptive. Compare expected cost with a worst-case criterion and explain what stakeholder preference each expresses.

Part IV

Batch Scheduling

10 An idealized vessel schedule

Problem 10 | Batch Production with Sequence-Dependent Cleaning | Level 1: Guided problem

10.1 Problem Statement

Four nonpreemptive batches must be processed on one vessel. The vessel is clean and available at time zero. A release time is the earliest processing start, not the earliest cleaning start. Cleaning may occur while the next batch awaits release. Initial and final cleaning times are zero.

Job	Processing time (h)	Release (h)	Due time (h)	Tardiness charge (USD/h)
A1	2	0	6	4
A2	3	2	12	2
B1	2	1	7	5
C1	4	0	14	1

Cleaning times depend on product family, denoted by the first letter of the job name:

From / to	A	B	C
A	0	1	2
B	3	0	1
C	1	2	0

All cleaning times are in hours. Normal shift length is 12 h. For the reference models, processing starts lie between 0 and 30 h and the final completion time between 0 and 40 h.

Your task. Ignore cleaning times. Minimize the time at which all four jobs finish while respecting release times. Due times are reported but are not restrictions in this first model.

10.1.1 Before you code

Calculate the total processing time as a lower bound. Can the jobs be ordered to attain it without waiting for a release?

10.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

10.2 Model Formulation

Let $x_{ij} = 1$ if job j immediately follows i, with dummy nodes `start` and `end`. Let s_j be processing start and C_{max} final completion. Each job has exactly one predecessor and one successor; `start` has one outgoing arc and `end` one incoming arc. There is no direct start-end arc.

For job-to-job arcs,

$$s_j \geq s_i + p_i + t_{ij} - M(1 - x_{ij}), \qquad s_j \geq r_j.$$

For job-to-end arcs, replace s_j by C_{max} and set cleaning to zero. Use $M = 50$ h. This is sufficient because starts are at most 30 h, processing at most 4 h, and cleaning at most 3 h; a deactivated inequality is redundant. Positive processing times rule out disconnected directed cycles: summing the time inequalities around a cycle would imply a strictly positive duration is at most zero.

Set $t_{ij} = 0$ for every arc and minimize C_{max}. Due times do not enter the objective, so a makespan-optimal sequence can still deliver some jobs late. Multiple optimal sequences are possible.

10.3 Pyomo Implementation

The code below is the model-building portion of `models/scheduling.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.scheduling 1
```

```
"""Single-vessel sequencing with release times and directed cleaning."""

import itertools
import pyomo.environ as pyo
from models.common import value

JOBS = ["A1", "A2", "B1", "C1"]
DURATION = {"A1": 2, "A2": 3, "B1": 2, "C1": 4}  # h
RELEASE = {"A1": 0, "A2": 2, "B1": 1, "C1": 0}
DUE = {"A1": 6, "A2": 12, "B1": 7, "C1": 14}
WEIGHT = {"A1": 4, "A2": 2, "B1": 5, "C1": 1}  # USD/(h of tardiness)
CLEAN = {
    ("A", "B"): 1,
    ("A", "C"): 2,
    ("B", "A"): 3,
    ("B", "C"): 1,
    ("C", "A"): 1,
    ("C", "B"): 2,
}


def cleaning(i, j, level):
    return 0 if level == 1 or i == "start" or j == "end" else CLEAN.get((i[0], j[0]), 0)


def build(level):
    arcs = [
        (i, j)
        for i in ["start"] + JOBS
        for j in JOBS + ["end"]
        if i != j and (i, j) != ("start", "end")
    ]
    m = pyo.ConcreteModel(name="Batch vessel schedule")
    m.J = pyo.Set(initialize=JOBS)
    m.A = pyo.Set(initialize=arcs, dimen=2)
    m.x = pyo.Var(m.A, domain=pyo.Binary)
    m.s = pyo.Var(m.J, bounds=(0, 30))
```

```
    m.finish = pyo.Var(bounds=(0, 40))
    m.out = pyo.Constraint(
        ["start"] + JOBS,
        rule=lambda m, i: sum(m.x[i, j] for a, j in m.A if a == i) == 1,
    )
    m.inc = pyo.Constraint(
        JOBS + ["end"], rule=lambda m, j: sum(m.x[i, j] for i, b in m.A if b == j) == 1
    )
    m.release = pyo.Constraint(m.J, rule=lambda m, j: m.s[j] >= RELEASE[j])
    m.time = pyo.ConstraintList()
    for i, j in arcs:
        if i == "start":
            continue
        target = m.finish if j == "end" else m.s[j]
        m.time.add(
            target
            >= m.s[i] + DURATION[i] + cleaning(i, j, level) - 50 * (1 - m.x[i, j])
        )
    m.tardy = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.late = pyo.Constraint(
        m.J, rule=lambda m, j: m.tardy[j] >= m.s[j] + DURATION[j] - DUE[j]
    )
    m.overtime = pyo.Var(domain=pyo.NonNegativeReals)
    m.over = pyo.Constraint(expr=m.overtime >= m.finish - 12)
    objective = (
        m.finish
        if level < 3
        else sum(WEIGHT[j] * m.tardy[j] for j in m.J) + 0.5 * m.overtime
    )
    m.obj = pyo.Objective(expr=objective)
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(1))
```

10.4 Optimal Solution

The reference objective is **11.0000 h**. This value is optimal for the explicitly stated model and data.

Job	Start (h)	Finish (h)	Tardiness (h)
A1	0.0000	2.0000	0.0000
C1	2.0000	6.0000	0.0000
B1	6.0000	8.0000	1.0000
A2	8.0000	11.0000	0.0000

Quantity	Value
Sequence	A1 > C1 > B1 > A2
Makespan (h)	11.0000
Cleaning (h)	0.0000

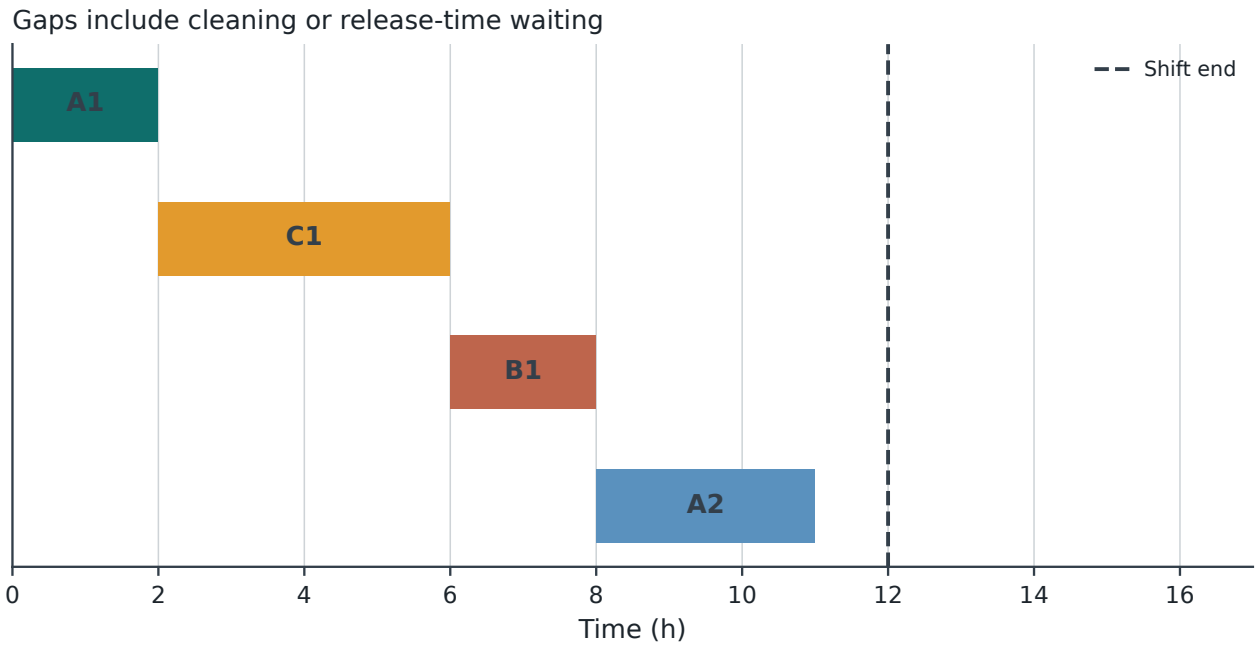

Figure 10.1: Batch Production with Sequence-Dependent Cleaning: reference solution for level 1.

10.4.1 Check your solution

Enumerate all 24 permutations. For each one, place every job at its earliest feasible start and calculate makespan. Check that processing intervals do not overlap.

The automated residual, bound, and integrality audit passed with a maximum violation of 1.00e-06 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

10.5 Brief Discussion

One optimal sequence is A1 > C1 > B1 > A2, with makespan 11.00 h and 0.00 h of cleaning. The makespan is 0.00 h above the idealized no-cleaning lower bound. Equivalent optimal sequences may differ across solver versions; the objective and feasibility checks are the comparison criteria.

10.5.1 Extension to investigate

Increase the release time of A2 to 10 h. Determine whether idle time becomes unavoidable and identify a sequence that attains the new lower bound.

11 Cleaning changes the preferred sequence

Problem 11 | Batch Production with Sequence-Dependent Cleaning | Level 2: Model extension

11.1 Problem Statement

Four nonpreemptive batches must be processed on one vessel. The vessel is clean and available at time zero. A release time is the earliest processing start, not the earliest cleaning start. Cleaning may occur while the next batch awaits release. Initial and final cleaning times are zero.

Job	Processing time (h)	Release (h)	Due time (h)	Tardiness charge (USD/h)
A1	2	0	6	4
A2	3	2	12	2
B1	2	1	7	5
C1	4	0	14	1

Cleaning times depend on product family, denoted by the first letter of the job name:

From / to	A	B	C
A	0	1	2
B	3	0	1
C	1	2	0

All cleaning times are in hours. Normal shift length is 12 h. For the reference models, processing starts lie between 0 and 30 h and the final completion time between 0 and 40 h.

Your task. Include the directed cleaning-time matrix and again minimize makespan. Compare the result with the idealized schedule.

11.1.1 Before you code

Is the setup between A and B symmetric? Why must cleaning be charged only between immediate neighbors rather than every ordered pair?

11.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

11.2 Model Formulation

Let $x_{ij} = 1$ if job j immediately follows i, with dummy nodes `start` and `end`. Let s_j be processing start and C_{max} final completion. Each job has exactly one predecessor and one successor; `start` has one outgoing arc and `end` one incoming arc. There is no direct start-end arc.

For job-to-job arcs,

$$s_j \geq s_i + p_i + t_{ij} - M(1 - x_{ij}), \qquad s_j \geq r_j.$$

For job-to-end arcs, replace s_j by C_{max} and set cleaning to zero. Use $M = 50$ h. This is sufficient because starts are at most 30 h, processing at most 4 h, and cleaning at most 3 h; a deactivated inequality is redundant. Positive processing times rule out disconnected directed cycles: summing the time inequalities around a cycle would imply a strictly positive duration is at most zero.

Retain the time inequalities using the specified t_{ij}. The arc formulation is essential: a pairwise before/after indicator alone does not identify which batches are adjacent. The reference objective remains $\min C_{max}$, so late deliveries are still not penalized.

11.3 Pyomo Implementation

The code below is the model-building portion of `models/scheduling.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.scheduling 2
```

```
"""Single-vessel sequencing with release times and directed cleaning."""

import itertools
import pyomo.environ as pyo
from models.common import value

JOBS = ["A1", "A2", "B1", "C1"]
DURATION = {"A1": 2, "A2": 3, "B1": 2, "C1": 4}  # h
RELEASE = {"A1": 0, "A2": 2, "B1": 1, "C1": 0}
DUE = {"A1": 6, "A2": 12, "B1": 7, "C1": 14}
WEIGHT = {"A1": 4, "A2": 2, "B1": 5, "C1": 1}  # USD/(h of tardiness)
CLEAN = {
    ("A", "B"): 1,
    ("A", "C"): 2,
    ("B", "A"): 3,
    ("B", "C"): 1,
    ("C", "A"): 1,
    ("C", "B"): 2,
}


def cleaning(i, j, level):
    return 0 if level == 1 or i == "start" or j == "end" else CLEAN.get((i[0], j[0]), 0)


def build(level):
    arcs = [
        (i, j)
        for i in ["start"] + JOBS
        for j in JOBS + ["end"]
        if i != j and (i, j) != ("start", "end")
    ]
    m = pyo.ConcreteModel(name="Batch vessel schedule")
    m.J = pyo.Set(initialize=JOBS)
    m.A = pyo.Set(initialize=arcs, dimen=2)
    m.x = pyo.Var(m.A, domain=pyo.Binary)
```

```
    m.s = pyo.Var(m.J, bounds=(0, 30))
    m.finish = pyo.Var(bounds=(0, 40))
    m.out = pyo.Constraint(
        ["start"] + JOBS,
        rule=lambda m, i: sum(m.x[i, j] for a, j in m.A if a == i) == 1,
    )
    m.inc = pyo.Constraint(
        JOBS + ["end"], rule=lambda m, j: sum(m.x[i, j] for i, b in m.A if b == j) == 1
    )
    m.release = pyo.Constraint(m.J, rule=lambda m, j: m.s[j] >= RELEASE[j])
    m.time = pyo.ConstraintList()
    for i, j in arcs:
        if i == "start":
            continue
        target = m.finish if j == "end" else m.s[j]
        m.time.add(
            target
            >= m.s[i] + DURATION[i] + cleaning(i, j, level) - 50 * (1 - m.x[i, j])
        )
    m.tardy = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.late = pyo.Constraint(
        m.J, rule=lambda m, j: m.tardy[j] >= m.s[j] + DURATION[j] - DUE[j]
    )
    m.overtime = pyo.Var(domain=pyo.NonNegativeReals)
    m.over = pyo.Constraint(expr=m.overtime >= m.finish - 12)
    objective = (
        m.finish
        if level < 3
        else sum(WEIGHT[j] * m.tardy[j] for j in m.J) + 0.5 * m.overtime
    )
    m.obj = pyo.Objective(expr=objective)
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(2))
```

11.4 Optimal Solution

The reference objective is **13.0000 h**. This value is optimal for the explicitly stated model and data.

Job	Start (h)	Finish (h)	Tardiness (h)
A1	-0.0000	2.0000	0.0000
A2	2.0000	5.0000	0.0000
B1	6.0000	8.0000	1.0000
C1	9.0000	13.0000	0.0000

Quantity	Value
Sequence	A1 > A2 > B1 > C1
Makespan (h)	13.0000
Cleaning (h)	2.0000

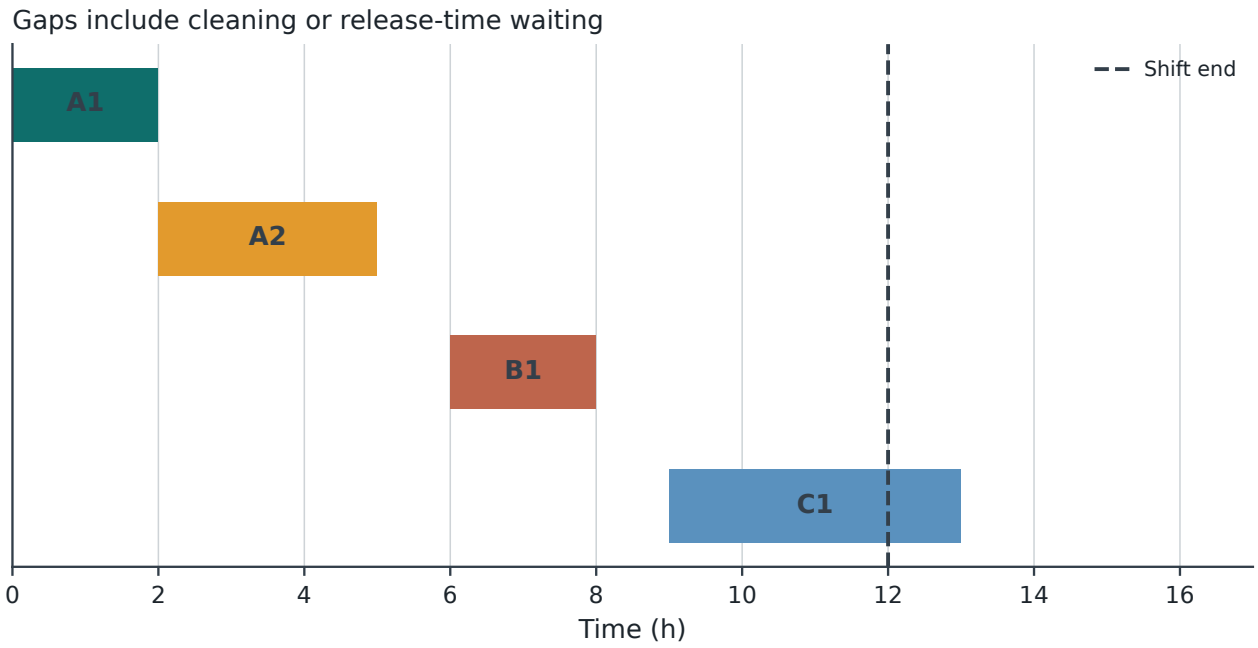

Figure 11.1: Batch Production with Sequence-Dependent Cleaning: reference solution for level 2.

11.4.1 Check your solution

Independently total processing, cleaning, and idle time on the chosen path. Compare with exhaustive permutation enumeration.

The automated residual, bound, and integrality audit passed with a maximum violation of 1.69e-14 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

11.5 Brief Discussion

One optimal sequence is A1 > A2 > B1 > C1, with makespan 13.00 h and 2.00 h of cleaning. The makespan is 2.00 h above the idealized no-cleaning lower bound. Equivalent optimal sequences may differ across solver versions; the objective and feasibility checks are the comparison criteria.

11.5.1 Extension to investigate

Add a restriction forbidding a direct transition from B to A. Determine whether the optimal makespan changes.

12 Balancing late deliveries and overtime

Problem 12 | Batch Production with Sequence-Dependent Cleaning | Level 3: Open-ended challenge

12.1 Problem Statement

Four nonpreemptive batches must be processed on one vessel. The vessel is clean and available at time zero. A release time is the earliest processing start, not the earliest cleaning start. Cleaning may occur while the next batch awaits release. Initial and final cleaning times are zero.

Job	Processing time (h)	Release (h)	Due time (h)	Tardiness charge (USD/h)
A1	2	0	6	4
A2	3	2	12	2
B1	2	1	7	5
C1	4	0	14	1

Cleaning times depend on product family, denoted by the first letter of the job name:

From / to	A	B	C
A	0	1	2
B	3	0	1
C	1	2	0

All cleaning times are in hours. Normal shift length is 12 h. For the reference models, processing starts lie between 0 and 30 h and the final completion time between 0 and 40 h.

Your task. Use the cleaning matrix and due times. Minimize total weighted tardiness plus overtime charged at 0.5 USD/h beyond the 12 h shift. Recommend whether these penalty rates represent a defensible service policy, and propose an alternative if some due times are contractual.

12.1.1 Before you code

Can the least-cost schedule have a longer makespan than the fastest schedule? Which late delivery is most expensive per hour?

12.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

12.2 Model Formulation

Let $x_{ij} = 1$ if job j immediately follows i, with dummy nodes `start` and `end`. Let s_j be processing start and C_{max} final completion. Each job has exactly one predecessor and one successor; `start` has one outgoing arc and `end` one incoming arc. There is no direct start-end arc.

For job-to-job arcs,

$$s_j \geq s_i + p_i + t_{ij} - M(1 - x_{ij}), \qquad s_j \geq r_j.$$

For job-to-end arcs, replace s_j by C_{max} and set cleaning to zero. Use $M = 50$ h. This is sufficient because starts are at most 30 h, processing at most 4 h, and cleaning at most 3 h; a deactivated inequality is redundant. Positive processing times rule out disconnected directed cycles: summing the time inequalities around a cycle would imply a strictly positive duration is at most zero.

Introduce $T_j \geq 0$ and $O \geq 0$ with

$$T_j \geq s_j + p_j - d_j, \qquad O \geq C_{max} - 12.$$

Minimize

$$\sum_j w_j T_j + 0.5O.$$

Positive coefficients make tardiness and overtime tight at optimum. This is a monetary objective, not lexicographic optimization. A hard due date would instead require $s_j + p_j \leq d_j$ and could make the model infeasible.

12.3 Pyomo Implementation

The code below is the model-building portion of `models/scheduling.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.scheduling 3
```

```
"""Single-vessel sequencing with release times and directed cleaning."""

import itertools
import pyomo.environ as pyo
from models.common import value

JOBS = ["A1", "A2", "B1", "C1"]
DURATION = {"A1": 2, "A2": 3, "B1": 2, "C1": 4}  # h
RELEASE = {"A1": 0, "A2": 2, "B1": 1, "C1": 0}
DUE = {"A1": 6, "A2": 12, "B1": 7, "C1": 14}
WEIGHT = {"A1": 4, "A2": 2, "B1": 5, "C1": 1}  # USD/(h of tardiness)
CLEAN = {
    ("A", "B"): 1,
    ("A", "C"): 2,
    ("B", "A"): 3,
    ("B", "C"): 1,
    ("C", "A"): 1,
    ("C", "B"): 2,
}


def cleaning(i, j, level):
    return 0 if level == 1 or i == "start" or j == "end" else CLEAN.get((i[0], j[0]), 0)


def build(level):
    arcs = [
        (i, j)
        for i in ["start"] + JOBS
```

```
        for j in JOBS + ["end"]
        if i != j and (i, j) != ("start", "end")
    ]
    m = pyo.ConcreteModel(name="Batch vessel schedule")
    m.J = pyo.Set(initialize=JOBS)
    m.A = pyo.Set(initialize=arcs, dimen=2)
    m.x = pyo.Var(m.A, domain=pyo.Binary)
    m.s = pyo.Var(m.J, bounds=(0, 30))
    m.finish = pyo.Var(bounds=(0, 40))
    m.out = pyo.Constraint(
        ["start"] + JOBS,
        rule=lambda m, i: sum(m.x[i, j] for a, j in m.A if a == i) == 1,
    )
    m.inc = pyo.Constraint(
        JOBS + ["end"], rule=lambda m, j: sum(m.x[i, j] for i, b in m.A if b == j) == 1
    )
    m.release = pyo.Constraint(m.J, rule=lambda m, j: m.s[j] >= RELEASE[j])
    m.time = pyo.ConstraintList()
    for i, j in arcs:
        if i == "start":
            continue
        target = m.finish if j == "end" else m.s[j]
        m.time.add(
            target
            >= m.s[i] + DURATION[i] + cleaning(i, j, level) - 50 * (1 - m.x[i, j])
        )
    m.tardy = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.late = pyo.Constraint(
        m.J, rule=lambda m, j: m.tardy[j] >= m.s[j] + DURATION[j] - DUE[j]
    )
    m.overtime = pyo.Var(domain=pyo.NonNegativeReals)
    m.over = pyo.Constraint(expr=m.overtime >= m.finish - 12)
    objective = (
        m.finish
        if level < 3
        else sum(WEIGHT[j] * m.tardy[j] for j in m.J) + 0.5 * m.overtime
    )
    m.obj = pyo.Objective(expr=objective)
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(3))
```

12.4 Optimal Solution

The reference objective is **5.0000 USD**. This value is optimal for the explicitly stated model and data.

Job	Start (h)	Finish (h)	Tardiness (h)
A1	0.0000	2.0000	0.0000
B1	3.0000	5.0000	0.0000
C1	6.0000	10.0000	0.0000
A2	11.0000	14.0000	2.0000

Quantity	Value
Sequence	A1 > B1 > C1 > A2

Quantity	Value
Makespan (h)	14.0000
Cleaning (h)	3.0000

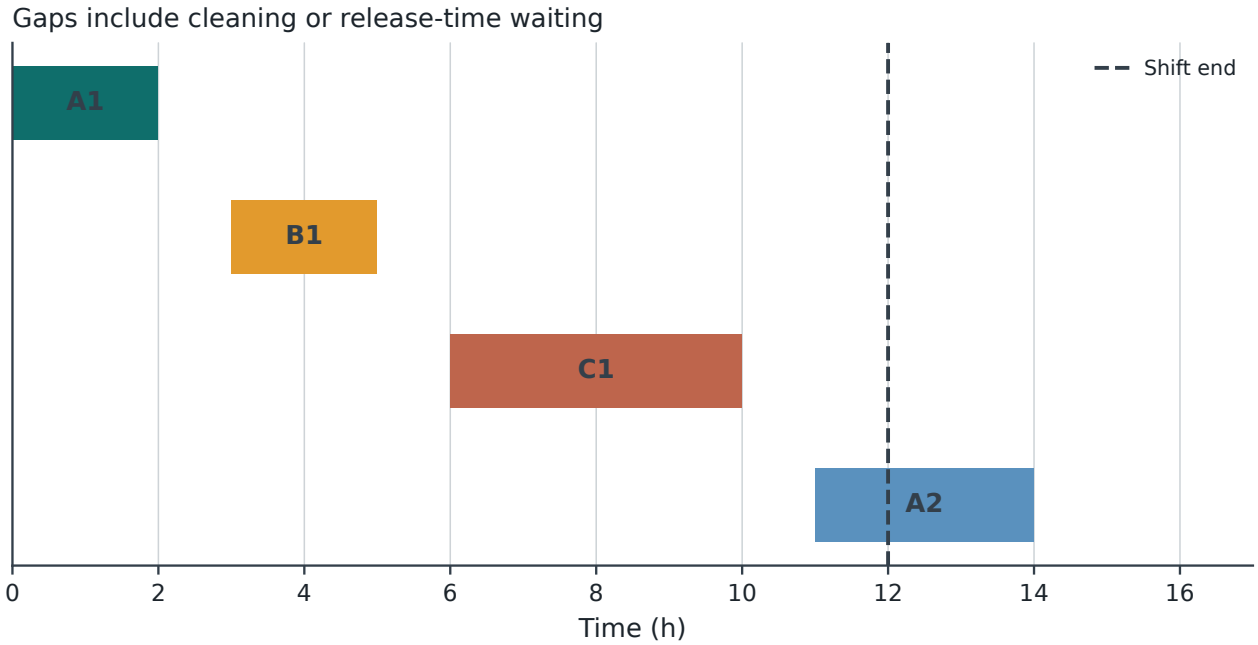

Figure 12.1: Batch Production with Sequence-Dependent Cleaning: reference solution for level 3.

12.4.1 Check your solution

Calculate tardiness from the actual finish times rather than trusting a displayed auxiliary variable. Independently enumerate all 24 sequences for the monetary objective.

The automated residual, bound, and integrality audit passed with a maximum violation of 3.55e-15 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

12.5 Brief Discussion

One optimal sequence is A1 > B1 > C1 > A2, with makespan 14.00 h and 3.00 h of cleaning. Total tardiness and overtime cost is 5.00 USD. The cost-optimal schedule need not be the fastest schedule. Equivalent optimal sequences may differ across solver versions; the objective and feasibility checks are the comparison criteria.

12.5.1 Extension to investigate

Sweep the overtime charge over 0, 0.5, 2, and 10 USD/h. Discuss when service and makespan objectives agree and when they conflict.

Part V

Hydrogen and Electricity

13 Dispatch with a fixed tank

Problem 13 | Hydrogen Production and Storage | Level 1: Guided problem

13.1 Problem Statement

An electrolyzer operates over six four-hour periods, representing one day. Electricity use is constant at 0.05 MWh/kg of hydrogen. Power is limited to 4 MW. Initial hydrogen inventory is 50 kg, and final inventory must also be 50 kg. Storage has no leakage and no charging-rate restriction. Production and withdrawal rates are each uniform within a period. Inventory therefore varies linearly between period boundaries.

Period	Electricity price (USD/MWh)	Hydrogen demand (kg/period)
1	30	120
2	25	150
3	80	180
4	100	220
5	45	180
6	35	140

Capacity is enforced at every period boundary, including the 50 kg initial stock. Under the uniform-rate assumption, this also bounds inventory throughout each period. If deliveries are discrete or rates vary within a period, an additional inventory profile or shorter time steps is necessary for physical sizing. A daily capacity charge of 0.15 USD/(kg d) is used where investment is optimized. It is an assumed daily equivalent, not a full capital-cost model.

Your task. Fix the tank at 150 kg and minimize daily electricity cost. Meet every period demand without purchases from an external supplier.

13.1.1 Before you code

Which hours are cheapest? Why might the electrolyzer still need to operate in an expensive period despite cheap electricity earlier?

13.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

13.2 Model Formulation

Let P_{st} be power in MW, I_{st} end inventory in kg, and K capacity in kg. With $\Delta t = 4$ h and $e = 0.05$ MWh/kg,

$$I_{st} = I_{s,t-1} + \frac{\Delta t}{e} P_{st} - D_{st}, \qquad 0 \le P_{st} \le 4, \quad 0 \le I_{st} \le K.$$

Use $I_{s0} = 50$ before the first period and $I_{s6} = 50$ at the end. Total production must equal total demand. The electricity cost is $\sum_t c_t \Delta t P_{st}$ USD/d. Omitting Δt would understate cost by a factor of four.

Fix $K = 150$ kg and minimize electricity cost under the inventory balances. Initial inventory is not free net supply because the terminal condition returns the same amount. The LP does not impose a minimum operating load or a startup charge.

13.3 Pyomo Implementation

The code below is the model-building portion of `models/hydrogen.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.hydrogen 1
```

```
"""Daily hydrogen dispatch, tank design, and scenario-dependent operation."""

import pyomo.environ as pyo
from models.common import value

PRICE = [30, 25, 80, 100, 45, 35]  # USD/MWh
DEMAND = [120, 150, 180, 220, 180, 140]  # kg per four-hour period
DT = 4  # h
SPECIFIC_ENERGY = 0.05  # MWh/kg
TANK_CHARGE = 0.15  # USD/(kg capacity d)


def build(level):
    states = (
        {"nominal": (1, 1)} if level < 3 else {"low": (0.9, 0.5), "high": (1.15, 0.5)}
    )
    m = pyo.ConcreteModel(name="Hydrogen production and storage")
    m.S = pyo.Set(initialize=list(states))
    m.T = pyo.RangeSet(0, 5)
    m.k = pyo.Var(bounds=(50, 500))
    if level == 1:
        m.k.fix(150)
    m.power = pyo.Var(m.S, m.T, bounds=(0, 4))
    m.stock = pyo.Var(m.S, m.T, domain=pyo.NonNegativeReals)
    m.balance = pyo.Constraint(
        m.S,
        m.T,
        rule=lambda m, s, t: (
            m.stock[s, t]
            == (50 if t == 0 else m.stock[s, t - 1])
            + DT * m.power[s, t] / SPECIFIC_ENERGY
            - DEMAND[t] * states[s][0]
        ),
    )
    m.capacity = pyo.Constraint(m.S, m.T, rule=lambda m, s, t: m.stock[s, t] <= m.k)
    m.terminal = pyo.Constraint(m.S, rule=lambda m, s: m.stock[s, 5] == 50)
    if level == 3:
        m.on = pyo.Var(m.S, m.T, domain=pyo.Binary)
        m.start = pyo.Var(m.S, m.T, domain=pyo.Binary)
        m.upper = pyo.Constraint(
```

```
            m.S, m.T, rule=lambda m, s, t: m.power[s, t] <= 4 * m.on[s, t]
        )
        m.lower = pyo.Constraint(
            m.S, m.T, rule=lambda m, s, t: m.power[s, t] >= m.on[s, t]
        )
        m.startup = pyo.Constraint(
            m.S,
            m.T,
            rule=lambda m, s, t: (
                m.start[s, t] >= m.on[s, t] - (0 if t == 0 else m.on[s, t - 1])
            ),
        )
    fixed = 0 if level == 1 else TANK_CHARGE * m.k
    m.obj = pyo.Objective(
        expr=fixed
        + sum(
            states[s][1]
            * sum(
                DT * PRICE[t] * m.power[s, t]
                + (10 * m.start[s, t] if level == 3 else 0)
                for t in m.T
            )
            for s in m.S
        )
    )
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(1))
```

13.4 Optimal Solution

The reference objective is **2,287.5000 USD/d**. This value is optimal for the explicitly stated model and data.

State	Period	Power (MW)	Demand (kg)	End stock (kg)
nominal	1	0.8750	120	0.0000
nominal	2	3.7500	150	150.0000
nominal	3	2.2500	180	150.0000
nominal	4	0.8750	220	0.0000
nominal	5	2.2500	180	0.0000
nominal	6	2.3750	140	50.0000

Quantity	Value
Tank capacity (kg)	150.0000
Expected electricity (MWh/d)	49.5000

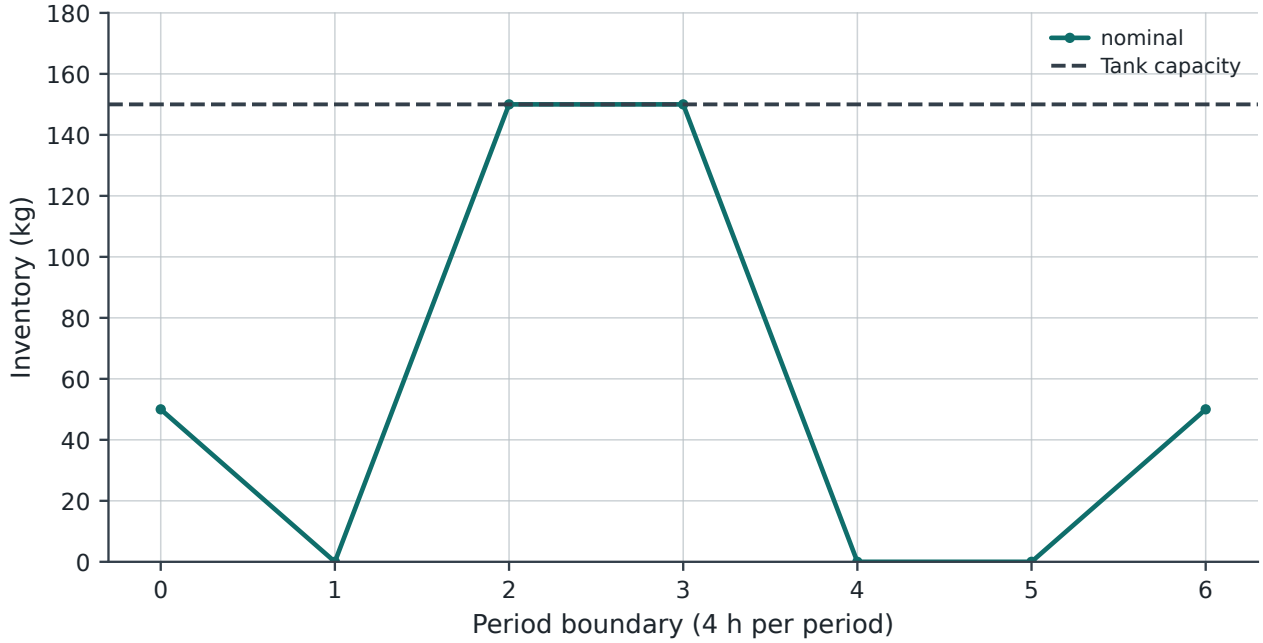

Figure 13.1: Hydrogen Production and Storage: reference solution for level 1.

13.4.1 Check your solution

Check daily production equals 990 kg and electricity use equals 49.5 MWh. Verify nonnegative inventory in all periods and the 50 kg terminal stock.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

13.5 Brief Discussion

Daily electricity costs 2287.50 USD. Total energy is 49.50 MWh: the storage benefit comes from purchase timing, not reduced energy use.

13.5.1 Extension to investigate

Remove the terminal condition and quantify the apparent savings. Explain why consuming starting stock is not a repeatable daily operating policy.

14 Choosing tank capacity

Problem 14 | Hydrogen Production and Storage | Level 2: Model extension

14.1 Problem Statement

An electrolyzer operates over six four-hour periods, representing one day. Electricity use is constant at 0.05 MWh/kg of hydrogen. Power is limited to 4 MW. Initial hydrogen inventory is 50 kg, and final inventory must also be 50 kg. Storage has no leakage and no charging-rate restriction. Production and withdrawal rates are each uniform within a period. Inventory therefore varies linearly between period boundaries.

Period	Electricity price (USD/MWh)	Hydrogen demand (kg/period)
1	30	120
2	25	150
3	80	180
4	100	220
5	45	180
6	35	140

Capacity is enforced at every period boundary, including the 50 kg initial stock. Under the uniform-rate assumption, this also bounds inventory throughout each period. If deliveries are discrete or rates vary within a period, an additional inventory profile or shorter time steps is necessary for physical sizing. A daily capacity charge of 0.15 USD/(kg d) is used where investment is optimized. It is an assumed daily equivalent, not a full capital-cost model.

Your task. Choose $50 \leq K \leq 500$ kg and minimize electricity cost plus the daily capacity charge. Compare with the 150 kg tank, charging that fixed tank the same daily capacity rate for a fair total-cost comparison.

14.1.1 Before you code

Can additional storage change total daily electricity use in this constant-efficiency model? If not, where do the savings come from?

14.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

14.2 Model Formulation

Let P_{st} be power in MW, I_{st} end inventory in kg, and K capacity in kg. With $\Delta t = 4$ h and $e = 0.05$ MWh/kg,

$$I_{st} = I_{s,t-1} + \frac{\Delta t}{e} P_{st} - D_{st}, \qquad 0 \le P_{st} \le 4, \quad 0 \le I_{st} \le K.$$

Use $I_{s0} = 50$ before the first period and $I_{s6} = 50$ at the end. Total production must equal total demand. The electricity cost is $\sum_t c_t \Delta t P_{st}$ USD/d. Omitting Δt would understate cost by a factor of four.

Solve

$$\min C = 0.15K + \sum_t c_t \Delta t P_t.$$

The lower bound of 50 kg accommodates initial and final stock. This LP values the timing of production, not a change in thermodynamic efficiency. The comparison with Problem 13 must add 0.15(150) USD/d to its electricity-only objective.

14.3 Pyomo Implementation

The code below is the model-building portion of `models/hydrogen.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.hydrogen 2
```

```
"""Daily hydrogen dispatch, tank design, and scenario-dependent operation."""

import pyomo.environ as pyo
from models.common import value

PRICE = [30, 25, 80, 100, 45, 35]  # USD/MWh
DEMAND = [120, 150, 180, 220, 180, 140]  # kg per four-hour period
DT = 4  # h
SPECIFIC_ENERGY = 0.05  # MWh/kg
TANK_CHARGE = 0.15  # USD/(kg capacity d)


def build(level):
    states = (
        {"nominal": (1, 1)} if level < 3 else {"low": (0.9, 0.5), "high": (1.15, 0.5)}
    )
    m = pyo.ConcreteModel(name="Hydrogen production and storage")
    m.S = pyo.Set(initialize=list(states))
    m.T = pyo.RangeSet(0, 5)
    m.k = pyo.Var(bounds=(50, 500))
    if level == 1:
        m.k.fix(150)
    m.power = pyo.Var(m.S, m.T, bounds=(0, 4))
    m.stock = pyo.Var(m.S, m.T, domain=pyo.NonNegativeReals)
    m.balance = pyo.Constraint(
        m.S,
        m.T,
        rule=lambda m, s, t: (
            m.stock[s, t]
            == (50 if t == 0 else m.stock[s, t - 1])
            + DT * m.power[s, t] / SPECIFIC_ENERGY
            - DEMAND[t] * states[s][0]
        ),
    )
    m.capacity = pyo.Constraint(m.S, m.T, rule=lambda m, s, t: m.stock[s, t] <= m.k)
    m.terminal = pyo.Constraint(m.S, rule=lambda m, s: m.stock[s, 5] == 50)
    if level == 3:
```

```
        m.on = pyo.Var(m.S, m.T, domain=pyo.Binary)
        m.start = pyo.Var(m.S, m.T, domain=pyo.Binary)
        m.upper = pyo.Constraint(
            m.S, m.T, rule=lambda m, s, t: m.power[s, t] <= 4 * m.on[s, t]
        )
        m.lower = pyo.Constraint(
            m.S, m.T, rule=lambda m, s, t: m.power[s, t] >= m.on[s, t]
        )
        m.startup = pyo.Constraint(
            m.S,
            m.T,
            rule=lambda m, s, t: (
                m.start[s, t] >= m.on[s, t] - (0 if t == 0 else m.on[s, t - 1])
            ),
        )
    fixed = 0 if level == 1 else TANK_CHARGE * m.k
    m.obj = pyo.Objective(
        expr=fixed
        + sum(
            states[s][1]
            * sum(
                DT * PRICE[t] * m.power[s, t]
                + (10 * m.start[s, t] if level == 3 else 0)
                for t in m.T
            )
            for s in m.S
        )
    )
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(2))
```

14.4 Optimal Solution

The reference objective is **1,635.5000 USD/d**. This value is optimal for the explicitly stated model and data.

State	Period	Power (MW)	Demand (kg)	End stock (kg)
nominal	1	4.0000	120	250.0000
nominal	2	4.0000	150	420.0000
nominal	3	0.0000	180	240.0000
nominal	4	0.0000	220	20.0000
nominal	5	2.0000	180	0.0000
nominal	6	2.3750	140	50.0000

Quantity	Value
Tank capacity (kg)	420.0000
Expected electricity (MWh/d)	49.5000

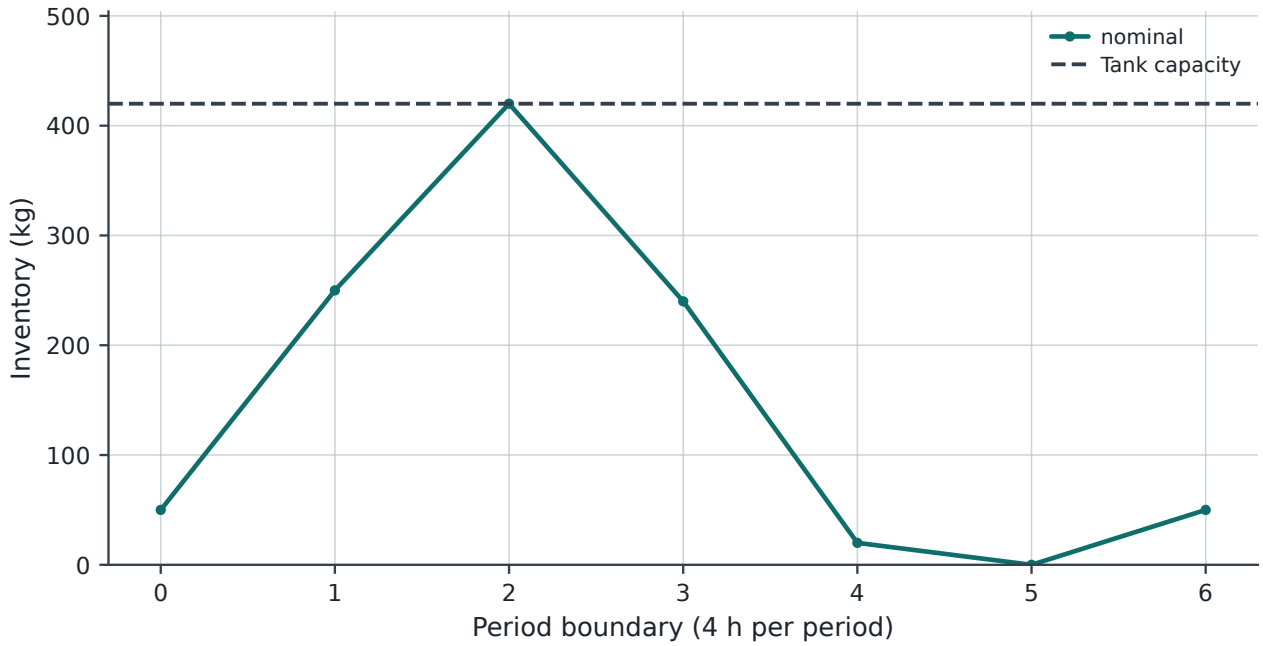

Figure 14.1: Hydrogen Production and Storage: reference solution for level 2.

14.4.1 Check your solution

Check the chosen tank is large enough for every reported end stock. Re-solve with fixed capacity values and verify that electricity savings eventually cease as capacity grows.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

14.4.2 Fixed-capacity experiment

capacity_kg	total_cost	electricity_cost
50.0000	2670.0000	2662.5000
100.0000	2490.0000	2475.0000
150.0000	2310.0000	2287.5000
200.0000	2137.5000	2107.5000
300.0000	1882.5000	1837.5000
400.0000	1647.5000	1587.5000
420.0000	1635.5000	1572.5000
450.0000	1640.0000	1572.5000
500.0000	1647.5000	1572.5000

14.5 Brief Discussion

The selected tank is 420.0 kg. Total daily cost is 1635.50 USD versus 2310.00 USD for the fixed 150 kg tank with the same ownership charge, saving 674.50 USD/d (29.20%). This is a daily-equivalent planning result under uniform within-period flows.

14.5.1 Extension to investigate

Estimate the break-even daily capacity charge for increasing the tank beyond 150 kg. Describe how finer time resolution might alter the design.

15 Design before demand is known

Problem 15 | Hydrogen Production and Storage | Level 3: Open-ended challenge

15.1 Problem Statement

An electrolyzer operates over six four-hour periods, representing one day. Electricity use is constant at 0.05 MWh/kg of hydrogen. Power is limited to 4 MW. Initial hydrogen inventory is 50 kg, and final inventory must also be 50 kg. Storage has no leakage and no charging-rate restriction. Production and withdrawal rates are each uniform within a period. Inventory therefore varies linearly between period boundaries.

Period	Electricity price (USD/MWh)	Hydrogen demand (kg/period)
1	30	120
2	25	150
3	80	180
4	100	220
5	45	180
6	35	140

Capacity is enforced at every period boundary, including the 50 kg initial stock. Under the uniform-rate assumption, this also bounds inventory throughout each period. If deliveries are discrete or rates vary within a period, an additional inventory profile or shorter time steps is necessary for physical sizing. A daily capacity charge of 0.15 USD/(kg d) is used where investment is optimized. It is an assumed daily equivalent, not a full capital-cost model.

Your task. Keep one shared tank capacity. Daily demand is either 0.9 or 1.15 times the table, each with probability 0.5. The demand state is fully observed before the first dispatch decision. In either state, the electrolyzer is initially off, consumes at least 1 MW when on, and incurs 10 USD per startup. Minimize capacity cost plus expected electricity and startup cost.

15.1.1 Before you code

Why may dispatch depend on the state while tank capacity may not? Would the same model be valid if the demand state were revealed only halfway through the day?

15.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

15.2 Model Formulation

Let P_{st} be power in MW, I_{st} end inventory in kg, and K capacity in kg. With $\Delta t = 4$ h and $e = 0.05$ MWh/kg,

$$I_{st} = I_{s,t-1} + \frac{\Delta t}{e} P_{st} - D_{st}, \qquad 0 \le P_{st} \le 4, \quad 0 \le I_{st} \le K.$$

Use $I_{s0} = 50$ before the first period and $I_{s6} = 50$ at the end. Total production must equal total demand. The electricity cost is $\sum_t c_t \Delta t P_{st}$ USD/d. Omitting Δt would understate cost by a factor of four.

Use state-specific binaries y_{st} for operation and b_{st} for startup:

$$y_{st} \le P_{st} \le 4 y_{st}, \qquad b_{st} \ge y_{st} - y_{s,t-1}, \quad y_{s0} = 0.$$

The minimum-load coefficient is 1 MW. Minimize

$$0.15K + \sum_s \pi_s \sum_t \left(c_t \Delta t P_{st} + 10 b_{st} \right).$$

The positive startup cost makes unnecessary startups unattractive. Capacity is shared across the two states; dispatch is recourse after full daily demand revelation. There is no minimum up/down time, ramp limit, or commitment carried to the next day. The terminal stock is restored separately in both states.

15.3 Pyomo Implementation

The code below is the model-building portion of `models/hydrogen.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.hydrogen 3
```

```
"""Daily hydrogen dispatch, tank design, and scenario-dependent operation."""

import pyomo.environ as pyo
from models.common import value

PRICE = [30, 25, 80, 100, 45, 35]  # USD/MWh
DEMAND = [120, 150, 180, 220, 180, 140]  # kg per four-hour period
DT = 4  # h
SPECIFIC_ENERGY = 0.05  # MWh/kg
TANK_CHARGE = 0.15  # USD/(kg capacity d)


def build(level):
    states = (
        {"nominal": (1, 1)} if level < 3 else {"low": (0.9, 0.5), "high": (1.15, 0.5)}
    )
    m = pyo.ConcreteModel(name="Hydrogen production and storage")
    m.S = pyo.Set(initialize=list(states))
    m.T = pyo.RangeSet(0, 5)
    m.k = pyo.Var(bounds=(50, 500))
    if level == 1:
        m.k.fix(150)
    m.power = pyo.Var(m.S, m.T, bounds=(0, 4))
    m.stock = pyo.Var(m.S, m.T, domain=pyo.NonNegativeReals)
    m.balance = pyo.Constraint(
        m.S,
        m.T,
        rule=lambda m, s, t: (
            m.stock[s, t]
            == (50 if t == 0 else m.stock[s, t - 1])
            + DT * m.power[s, t] / SPECIFIC_ENERGY
            - DEMAND[t] * states[s][0]
```

```
        ),
    )
    m.capacity = pyo.Constraint(m.S, m.T, rule=lambda m, s, t: m.stock[s, t] <= m.k)
    m.terminal = pyo.Constraint(m.S, rule=lambda m, s: m.stock[s, 5] == 50)
    if level == 3:
        m.on = pyo.Var(m.S, m.T, domain=pyo.Binary)
        m.start = pyo.Var(m.S, m.T, domain=pyo.Binary)
        m.upper = pyo.Constraint(
            m.S, m.T, rule=lambda m, s, t: m.power[s, t] <= 4 * m.on[s, t]
        )
        m.lower = pyo.Constraint(
            m.S, m.T, rule=lambda m, s, t: m.power[s, t] >= m.on[s, t]
        )
        m.startup = pyo.Constraint(
            m.S,
            m.T,
            rule=lambda m, s, t: (
                m.start[s, t] >= m.on[s, t] - (0 if t == 0 else m.on[s, t - 1])
            ),
        )
    fixed = 0 if level == 1 else TANK_CHARGE * m.k
    m.obj = pyo.Objective(
        expr=fixed
        + sum(
            states[s][1]
            * sum(
                DT * PRICE[t] * m.power[s, t]
                + (10 * m.start[s, t] if level == 3 else 0)
                for t in m.T
            )
            for s in m.S
        )
    )
    m._states = states
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(3))
```

15.4 Optimal Solution

The reference objective is **1,785.0500 USD/d**. This value is optimal for the explicitly stated model and data.

State	Period	Power (MW)	Demand (kg)	End stock (kg)
low	1	3.9375	108.0000	257.0000
low	2	4.0000	135.0000	442.0000
low	3	0.0000	162.0000	280.0000
low	4	-0.0000	198.0000	82.0000
low	5	1.0000	162.0000	0.0000
low	6	2.2000	126.0000	50.0000
high	1	4.0000	138.0000	232.0000
high	2	4.0000	172.5000	379.5000
high	3	1.0062	207.0000	253.0000
high	4	0.0000	253.0000	0.0000
high	5	2.5875	207.0000	0.0000
high	6	2.6375	161.0000	50.0000

State	Period	Power (MW)	Demand (kg)	End stock (kg)

Quantity	Value
Tank capacity (kg)	442.0000
Expected electricity (MWh/d)	50.7375

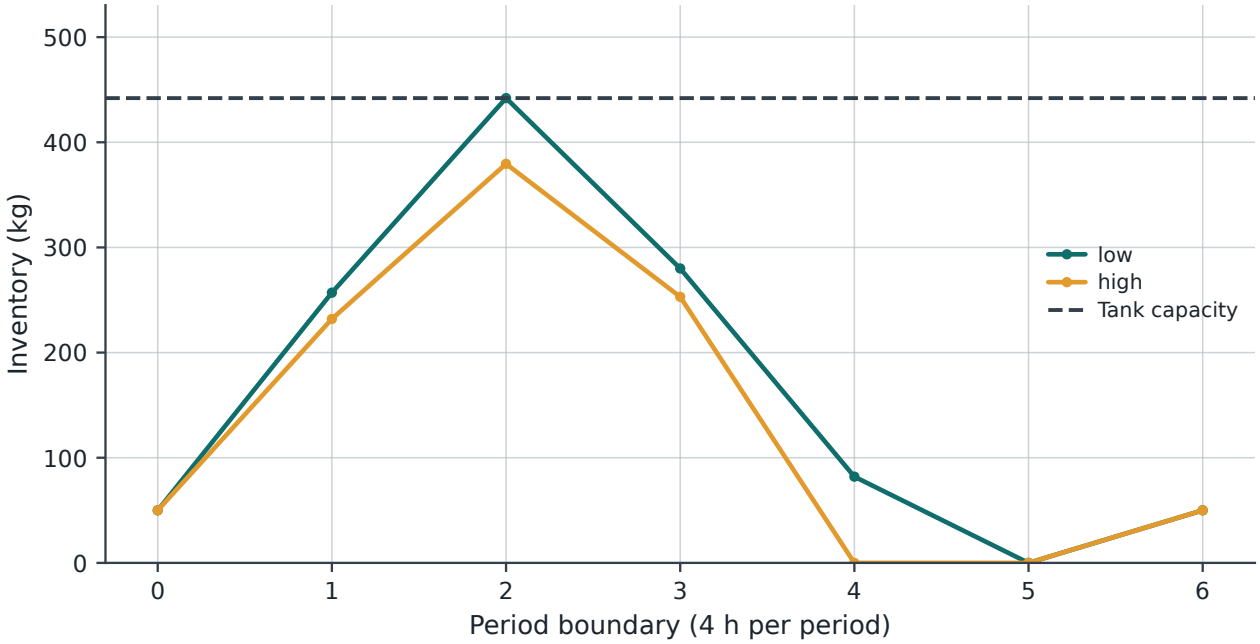

Figure 15.1: Hydrogen Production and Storage: reference solution for level 3.

15.4.1 Check your solution

Check power is either zero or between 1 and 4 MW. Check mass balance and terminal stock in each state. Count startups directly from the on/off sequence.

The automated residual, bound, and integrality audit passed with a maximum violation of 2.84e-14 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

15.5 Brief Discussion

The shared tank is 442.0 kg and expected daily cost is 1785.05 USD. Expected demand is 1.025 times the original profile, so the change from the deterministic case combines a higher mean demand, uncertainty, and commitment costs. It is not a pure value-of-uncertainty comparison.

15.5.1 Extension to investigate

Formulate the nonanticipativity constraints required if the state is learned after period 2. State which decisions must coincide before that time, and explain how uncertainty in earlier withdrawals affects the scenario definition.

Part VI

Water Reuse

16 Direct reuse without treatment

Problem 16 | Wastewater Treatment and Reuse | Level 1: Guided problem

16.1 Problem Statement

Two independent wastewater sources supply S1 = 60 m^3/h at 40 mg/L and S2 = 50 m^3/h at 120 mg/L of one contaminant. Users U1 and U2 need 50 and 40 m^3/h, with maximum inlet concentrations of 30 and 60 mg/L. Freshwater contains no contaminant and costs 1 USD/m^3. Unused source water is discharged at 0.10 USD/m^3. User outlet streams are outside the model boundary; this is a one-pass reuse allocation.

Treatment	Removal fraction	Capacity (m3/h)	Variable cost (USD/m3)	Hourly ownership charge (USD/h)
Moderate	0.75	40	0.25	3
Advanced	0.95	80	0.45	12

Treatment conserves water volume and transfers removed contaminant to a separate captured-residue stream. Residue handling is included in the assumed treatment cost. Each source remains segregated through treatment, or equivalently uses source-specific modules with a shared total hydraulic capacity. Fixed removal fractions do not depend on loading. These assumptions avoid a common mixed treatment inlet with an unknown concentration.

Your task. Allow direct reuse and freshwater only. Minimize freshwater and disposal costs while meeting both user quality limits.

16.1.1 Before you code

Can source S1 be sent directly to U1 without dilution? Why might some S2 still be useful to U2?

16.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

16.2 Model Formulation

Let $x_{ij} \geq 0$ be direct reuse, $y_{ijk} \geq 0$ treated reuse through technology k, $f_j \geq 0$ freshwater, and $d_i \geq 0$ discharge, all in m^3/h. A binary z_k selects a treatment option when required. With source supply S_i, user demand D_j, source concentration c_i, user limit L_j, and removal r_k,

$$\sum_j \left(x_{ij} + \sum_k y_{ijk} \right) + d_i = S_i,$$

$$\sum_i \left(x_{ij} + \sum_k y_{ijk} \right) + f_j = D_j,$$

$$\sum_i c_i \left[x_{ij} + \sum_k (1 - r_k) y_{ijk} \right] \le L_j D_j,$$

$$\sum_{i,j} y_{ijk} \le \overline{Q}_k z_k.$$

Concentration-times-flow products have units mg/L times m^3/h. Multiplying every term by 1000 gives mg/h, so the common factor cancels in the quality inequality. Concentrations are fixed input coefficients, which keeps this formulation linear.

Remove all treatment variables and minimize

$$C = \sum_j f_j + 0.10 \sum_i d_i.$$

The quality limits are concentration limits at fixed total user demand, so they can be written as linear contaminant-load inequalities. Mixing at each user is permitted, but no unknown intermediate pool feeds multiple users.

16.3 Pyomo Implementation

The code below is the model-building portion of `models/water.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.water 1
```

```
"""Segregated wastewater reuse with fixed-quality source streams."""

import pyomo.environ as pyo
from models.common import value

SUPPLY = {"S1": 60, "S2": 50}  # m3/h
NOMINAL = {"S1": 40, "S2": 120}  # mg/L
HIGH = {"S1": 60, "S2": 160}
DEMAND = {"U1": 50, "U2": 40}  # m3/h
LIMIT = {"U1": 30, "U2": 60}  # mg/L
TECH = {"moderate": (0.75, 40, 0.25, 3), "advanced": (0.95, 80, 0.45, 12)}
# removal fraction, hydraulic capacity m3/h, variable USD/m3, fixed USD/h


def build(level):
    conc = HIGH if level == 3 else NOMINAL
    techs = [] if level == 1 else ["moderate"] if level == 2 else list(TECH)
    m = pyo.ConcreteModel(name="Segregated water reuse")
    m.I = pyo.Set(initialize=list(SUPPLY))
    m.J = pyo.Set(initialize=list(DEMAND))
    m.K = pyo.Set(initialize=techs)
    m.x = pyo.Var(m.I, m.J, domain=pyo.NonNegativeReals)
    m.y = pyo.Var(m.I, m.J, m.K, domain=pyo.NonNegativeReals)
    m.f = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.d = pyo.Var(m.I, domain=pyo.NonNegativeReals)
    m.z = pyo.Var(m.K, domain=pyo.Binary)
    if level == 2:
        m.z["moderate"].fix(1)
    if level == 3:
        m.choose = pyo.Constraint(expr=sum(m.z[k] for k in m.K) <= 1)
    m.source = pyo.Constraint(
        m.I,
```

```
        rule=lambda m, i: (
            sum(m.x[i, j] + sum(m.y[i, j, k] for k in m.K) for j in m.J) + m.d[i]
            == SUPPLY[i]
        ),
    )
    m.sink = pyo.Constraint(
        m.J,
        rule=lambda m, j: (
            sum(m.x[i, j] + sum(m.y[i, j, k] for k in m.K) for i in m.I) + m.f[j]
            == DEMAND[j]
        ),
    )
    m.quality = pyo.Constraint(
        m.J,
        rule=lambda m, j: (
            sum(
                conc[i]
                * (m.x[i, j] + sum((1 - TECH[k][0]) * m.y[i, j, k] for k in m.K))
                for i in m.I
            )
            <= LIMIT[j] * DEMAND[j]
        ),
    )
    m.cap = pyo.Constraint(
        m.K,
        rule=lambda m, k: (
            sum(m.y[i, j, k] for i in m.I for j in m.J) <= TECH[k][1] * m.z[k]
        ),
    )
    m.obj = pyo.Objective(
        expr=sum(m.f[j] for j in m.J)
        + 0.1 * sum(m.d[i] for i in m.I)
        + sum(TECH[k][2] * m.y[i, j, k] for i in m.I for j in m.J for k in m.K)
        + (sum(TECH[k][3] * m.z[k] for k in m.K) if level == 3 else 0)
    )
    m._conc = conc
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(1))
```

16.4 Optimal Solution

The reference objective is **21.2500 USD/h**. This value is optimal for the explicitly stated model and data.

User	Fresh (m3/h)	Direct reuse (m3/h)	Treated reuse (m3/h)	Concentration (mg/L)
U1	12.5000	37.5000	0	30.0000
U2	5.0000	35.0000	0	60.0000

Quantity	Value
Freshwater (m3/h)	17.5000
Disposal (m3/h)	37.5000
Technology	none

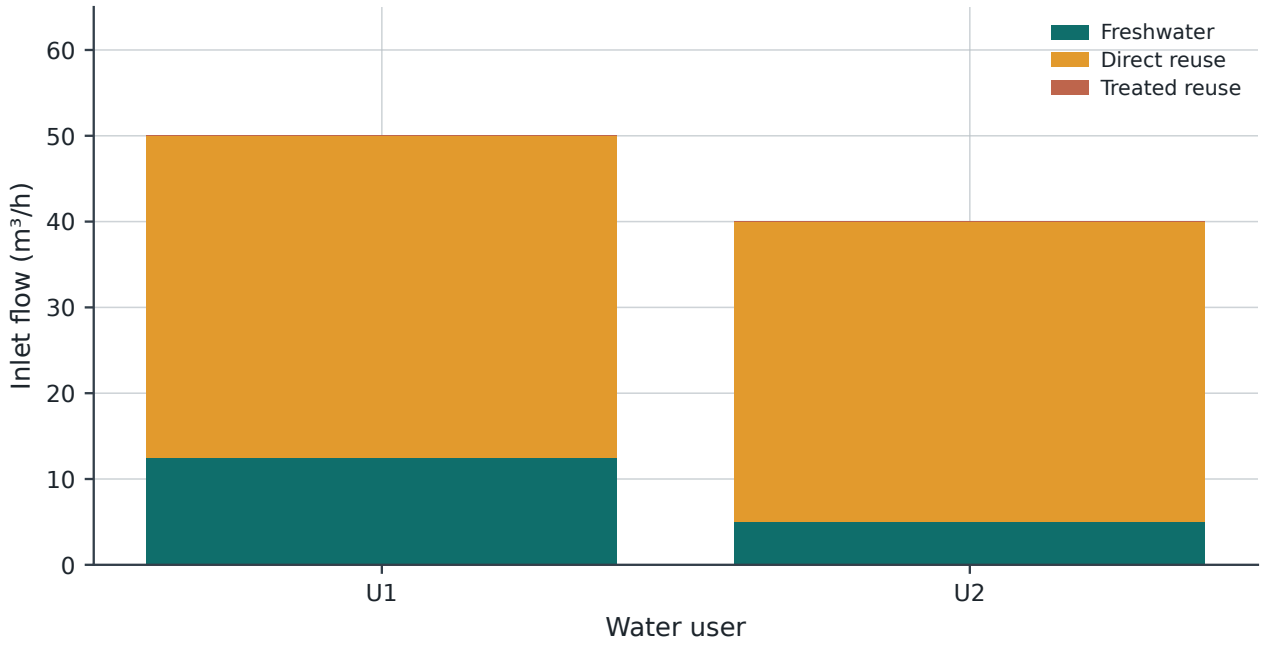

Figure 16.1: Wastewater Treatment and Reuse: reference solution for level 1.

16.4.1 Check your solution

Check the overall water balance $110 + \sum_j f_j = 90 + \sum_i d_i$. Independently calculate each user concentration from incoming contaminant load.

The automated residual, bound, and integrality audit passed with a maximum violation of 7.11e-15 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

16.5 Brief Discussion

The solution uses 17.500 m3/h of freshwater and discharges 37.500 m3/h, at 21.250 USD/h. Treatment choice: none. Contaminant removed by treatment must leave in the captured-residue stream. Fixed removal and segregated paths are substantive process assumptions, not just numerical conveniences.

16.5.1 Extension to investigate

Raise the U1 concentration limit from 30 to 40 mg/L and predict the freshwater reduction before solving.

17 Reuse with an existing treatment unit

Problem 17 | Wastewater Treatment and Reuse | Level 2: Model extension

17.1 Problem Statement

Two independent wastewater sources supply S1 = 60 m^3/h at 40 mg/L and S2 = 50 m^3/h at 120 mg/L of one contaminant. Users U1 and U2 need 50 and 40 m^3/h, with maximum inlet concentrations of 30 and 60 mg/L. Freshwater contains no contaminant and costs 1 USD/m^3. Unused source water is discharged at 0.10 USD/m^3. User outlet streams are outside the model boundary; this is a one-pass reuse allocation.

Treatment	Removal fraction	Capacity (m3/h)	Variable cost (USD/m3)	Hourly ownership charge (USD/h)
Moderate	0.75	40	0.25	3
Advanced	0.95	80	0.45	12

Treatment conserves water volume and transfers removed contaminant to a separate captured-residue stream. Residue handling is included in the assumed treatment cost. Each source remains segregated through treatment, or equivalently uses source-specific modules with a shared total hydraulic capacity. Fixed removal fractions do not depend on loading. These assumptions avoid a common mixed treatment inlet with an unknown concentration.

Your task. Add the moderate treatment option with capacity 40 m^3/h and removal fraction 0.75. Its ownership cost is sunk, so charge only variable treatment, freshwater, and disposal costs.

17.1.1 Before you code

For a given treated volume, which source yields a larger contaminant reduction? Does that automatically mean all treatment should serve that source?

17.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

17.2 Model Formulation

Let $x_{ij} \geq 0$ be direct reuse, $y_{ijk} \geq 0$ treated reuse through technology k, $f_j \geq 0$ freshwater, and $d_i \geq 0$ discharge, all in m^3/h. A binary z_k selects a treatment option when required. With source supply S_i, user demand D_j, source concentration c_i, user limit L_j, and removal r_k,

$$\sum_j \left(x_{ij} + \sum_k y_{ijk} \right) + d_i = S_i,$$

$$\sum_i \left(x_{ij} + \sum_k y_{ijk} \right) + f_j = D_j,$$

$$\sum_i c_i \left[x_{ij} + \sum_k (1 - r_k) y_{ijk} \right] \le L_j D_j,$$

$$\sum_{i,j} y_{ijk} \le \overline{Q}_k z_k.$$

Concentration-times-flow products have units mg/L times m^3/h. Multiplying every term by 1000 gives mg/h, so the common factor cancels in the quality inequality. Concentrations are fixed input coefficients, which keeps this formulation linear.

Fix $z_{moderate} = 1$ and minimize

$$C = \sum_j f_j + 0.10 \sum_i d_i + 0.25 \sum_{i,j} y_{ij,moderate}.$$

Removed contaminant is

$$R = \sum_{i,j} c_i r_{moderate} y_{ij,moderate}.$$

Include this captured load when checking the contaminant balance. A treatment model that reduces concentration while conserving water but has no contaminant outlet is physically incomplete.

17.3 Pyomo Implementation

The code below is the model-building portion of `models/water.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.water 2
```

```
"""Segregated wastewater reuse with fixed-quality source streams."""

import pyomo.environ as pyo
from models.common import value

SUPPLY = {"S1": 60, "S2": 50}  # m3/h
NOMINAL = {"S1": 40, "S2": 120}  # mg/L
HIGH = {"S1": 60, "S2": 160}
DEMAND = {"U1": 50, "U2": 40}  # m3/h
LIMIT = {"U1": 30, "U2": 60}  # mg/L
TECH = {"moderate": (0.75, 40, 0.25, 3), "advanced": (0.95, 80, 0.45, 12)}
# removal fraction, hydraulic capacity m3/h, variable USD/m3, fixed USD/h


def build(level):
    conc = HIGH if level == 3 else NOMINAL
    techs = [] if level == 1 else ["moderate"] if level == 2 else list(TECH)
    m = pyo.ConcreteModel(name="Segregated water reuse")
    m.I = pyo.Set(initialize=list(SUPPLY))
    m.J = pyo.Set(initialize=list(DEMAND))
    m.K = pyo.Set(initialize=techs)
    m.x = pyo.Var(m.I, m.J, domain=pyo.NonNegativeReals)
    m.y = pyo.Var(m.I, m.J, m.K, domain=pyo.NonNegativeReals)
    m.f = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.d = pyo.Var(m.I, domain=pyo.NonNegativeReals)
    m.z = pyo.Var(m.K, domain=pyo.Binary)
    if level == 2:
        m.z["moderate"].fix(1)
    if level == 3:
        m.choose = pyo.Constraint(expr=sum(m.z[k] for k in m.K) <= 1)
```

```
    m.source = pyo.Constraint(
        m.I,
        rule=lambda m, i: (
            sum(m.x[i, j] + sum(m.y[i, j, k] for k in m.K) for j in m.J) + m.d[i]
            == SUPPLY[i]
        ),
    )
    m.sink = pyo.Constraint(
        m.J,
        rule=lambda m, j: (
            sum(m.x[i, j] + sum(m.y[i, j, k] for k in m.K) for i in m.I) + m.f[j]
            == DEMAND[j]
        ),
    )
    m.quality = pyo.Constraint(
        m.J,
        rule=lambda m, j: (
            sum(
                conc[i]
                * (m.x[i, j] + sum((1 - TECH[k][0]) * m.y[i, j, k] for k in m.K))
                for i in m.I
            )
            <= LIMIT[j] * DEMAND[j]
        ),
    )
    m.cap = pyo.Constraint(
        m.K,
        rule=lambda m, k: (
            sum(m.y[i, j, k] for i in m.I for j in m.J) <= TECH[k][1] * m.z[k]
        ),
    )
    m.obj = pyo.Objective(
        expr=sum(m.f[j] for j in m.J)
        + 0.1 * sum(m.d[i] for i in m.I)
        + sum(TECH[k][2] * m.y[i, j, k] for i in m.I for j in m.J for k in m.K)
        + (sum(TECH[k][3] * m.z[k] for k in m.K) if level == 3 else 0)
    )
    m._conc = conc
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(2))
```

17.4 Optimal Solution

The reference objective is **9.5000 USD/h**. This value is optimal for the explicitly stated model and data.

User	Fresh (m3/h)	Direct reuse (m3/h)	Treated reuse (m3/h)	Concentration (mg/L)
U1	0.0000	20.0000	30.0000	30.0000
U2	0.0000	40.0000	0.0000	60.0000

Quantity	Value
Freshwater (m3/h)	0.0000
Disposal (m3/h)	20.0000
Technology	moderate

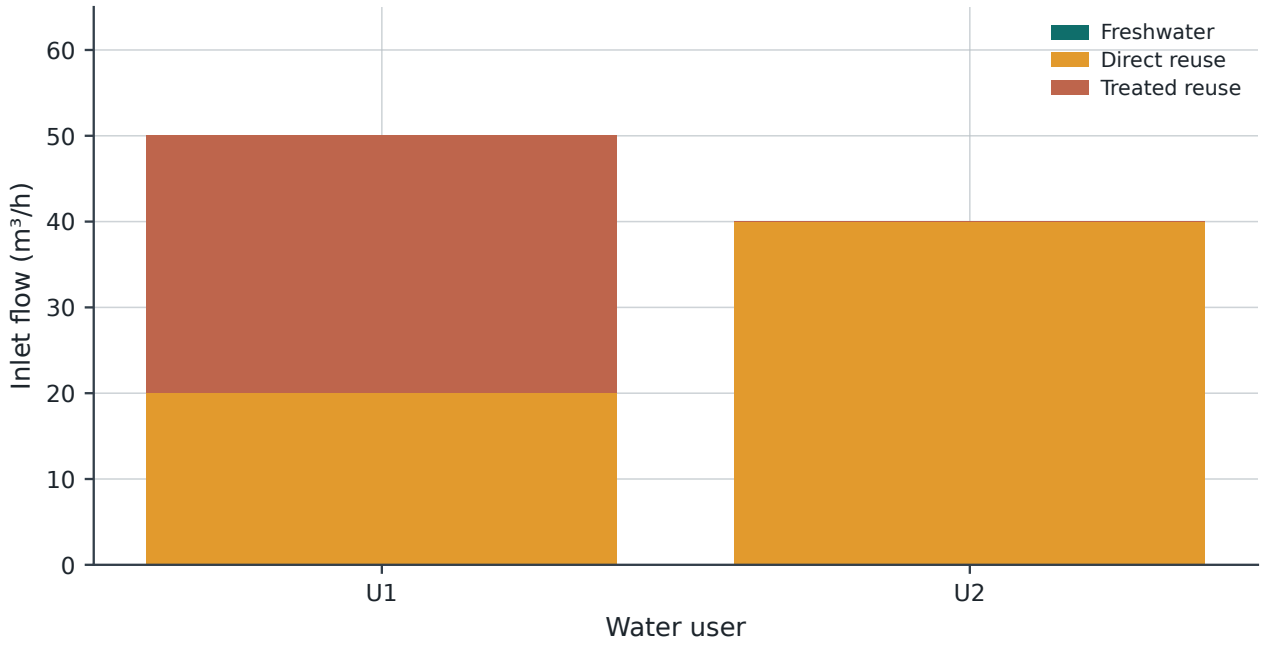

Figure 17.1: Wastewater Treatment and Reuse: reference solution for level 2.

17.4.1 Check your solution

Check source contaminant input equals load sent to users plus load in discharge plus captured contaminant. Convert the final quantities to kg/h using 0.001 kg per mg/L times m3.

The automated residual, bound, and integrality audit passed with a maximum violation of 7.11e-15 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

17.5 Brief Discussion

The solution uses 0.000 m3/h of freshwater and discharges 20.000 m3/h, at 9.500 USD/h. Treatment choice: moderate. Even when freshwater falls to zero, 20 m3/h must leave the system because source supply exceeds user demand. Fixed removal and segregated paths are substantive process assumptions, not just numerical conveniences.

17.5.1 Extension to investigate

Lower treatment removal to 0.50 and determine whether zero freshwater remains feasible at the same capacity.

18 Selecting treatment under composition uncertainty

Problem 18 | Wastewater Treatment and Reuse | Level 3: Open-ended challenge

18.1 Problem Statement

Two independent wastewater sources supply S1 = 60 m^3/h at 40 mg/L and S2 = 50 m^3/h at 120 mg/L of one contaminant. Users U1 and U2 need 50 and 40 m^3/h, with maximum inlet concentrations of 30 and 60 mg/L. Freshwater contains no contaminant and costs 1 USD/m^3. Unused source water is discharged at 0.10 USD/m^3. User outlet streams are outside the model boundary; this is a one-pass reuse allocation.

Treatment	Removal fraction	Capacity (m3/h)	Variable cost (USD/m3)	Hourly ownership charge (USD/h)
Moderate	0.75	40	0.25	3
Advanced	0.95	80	0.45	12

Treatment conserves water volume and transfers removed contaminant to a separate captured-residue stream. Residue handling is included in the assumed treatment cost. Each source remains segregated through treatment, or equivalently uses source-specific modules with a shared total hydraulic capacity. Fixed removal fractions do not depend on loading. These assumptions avoid a common mixed treatment inlet with an unknown concentration.

Your task. Choose at most one new treatment option, or none. Source concentrations can independently rise to 60 mg/L for S1 and 160 mg/L for S2. Require all user quality limits to hold throughout these intervals, using one fixed allocation. Minimize the deterministic cost including the selected hourly ownership charge. Explain when the segregated treatment assumption would be defensible.

18.1.1 Before you code

Which concentrations create the worst load? What nonlinear term appears if an intermediate mixed pool has both unknown flow and unknown concentration?

18.1.2 Deliverables

1. Define decision variables, units, objective, and constraints. State which data and assumptions are fixed.
2. Predict one feature of the solution and explain why it should occur.
3. Build and solve a Pyomo model. Check balances and bounds independently.
4. Interpret the result and investigate the follow-up question below. For an open-ended challenge, state and defend your chosen policy separately from the reference solution.

18.2 Model Formulation

Let $x_{ij} \geq 0$ be direct reuse, $y_{ijk} \geq 0$ treated reuse through technology k, $f_j \geq 0$ freshwater, and $d_i \geq 0$ discharge, all in m^3/h. A binary z_k selects a treatment option when required. With source supply S_i, user demand D_j, source concentration c_i, user limit L_j, and removal r_k,

$$\sum_j \left(x_{ij} + \sum_k y_{ijk} \right) + d_i = S_i,$$

$$\sum_i \left(x_{ij} + \sum_k y_{ijk} \right) + f_j = D_j,$$

$$\sum_i c_i \left[x_{ij} + \sum_k (1 - r_k) y_{ijk} \right] \leq L_j D_j,$$

$$\sum_{i,j} y_{ijk} \leq \overline{Q}_k z_k.$$

Concentration-times-flow products have units mg/L times m^3/h. Multiplying every term by 1000 gives mg/h, so the common factor cancels in the quality inequality. Concentrations are fixed input coefficients, which keeps this formulation linear.

Use the upper concentrations in every quality inequality, since all flows and residual fractions are nonnegative. Require $\sum_k z_k \leq 1$ and minimize

$$C = \sum_j f_j + 0.10 \sum_i d_i + \sum_{i,j,k} v_k y_{ijk} + \sum_k K_k z_k.$$

This is a robust MILP for fixed-quality, segregated treatment paths. In a general pooling model, a pool concentration c_P and outgoing flow q_{Pj} are both variables, producing the bilinear load $c_P q_{Pj}$. The present solution is not a global solution to that unrestricted nonlinear pooling problem. A separate nonlinear model and suitable solver would be required for it.

18.3 Pyomo Implementation

The code below is the model-building portion of `models/water.py`. The `level` argument selects this problem's assumptions. Run the complete module from the source bundle to reproduce the solution, including its independent checks:

```
~/.venvs/optim/bin/python -m models.water 3
```

```
"""Segregated wastewater reuse with fixed-quality source streams."""

import pyomo.environ as pyo
from models.common import value

SUPPLY = {"S1": 60, "S2": 50}  # m3/h
NOMINAL = {"S1": 40, "S2": 120}  # mg/L
HIGH = {"S1": 60, "S2": 160}
DEMAND = {"U1": 50, "U2": 40}  # m3/h
LIMIT = {"U1": 30, "U2": 60}  # mg/L
TECH = {"moderate": (0.75, 40, 0.25, 3), "advanced": (0.95, 80, 0.45, 12)}
# removal fraction, hydraulic capacity m3/h, variable USD/m3, fixed USD/h


def build(level):
    conc = HIGH if level == 3 else NOMINAL
    techs = [] if level == 1 else ["moderate"] if level == 2 else list(TECH)
    m = pyo.ConcreteModel(name="Segregated water reuse")
    m.I = pyo.Set(initialize=list(SUPPLY))
```

```
    m.J = pyo.Set(initialize=list(DEMAND))
    m.K = pyo.Set(initialize=techs)
    m.x = pyo.Var(m.I, m.J, domain=pyo.NonNegativeReals)
    m.y = pyo.Var(m.I, m.J, m.K, domain=pyo.NonNegativeReals)
    m.f = pyo.Var(m.J, domain=pyo.NonNegativeReals)
    m.d = pyo.Var(m.I, domain=pyo.NonNegativeReals)
    m.z = pyo.Var(m.K, domain=pyo.Binary)
    if level == 2:
        m.z["moderate"].fix(1)
    if level == 3:
        m.choose = pyo.Constraint(expr=sum(m.z[k] for k in m.K) <= 1)
    m.source = pyo.Constraint(
        m.I,
        rule=lambda m, i: (
            sum(m.x[i, j] + sum(m.y[i, j, k] for k in m.K) for j in m.J) + m.d[i]
            == SUPPLY[i]
        ),
    )
    m.sink = pyo.Constraint(
        m.J,
        rule=lambda m, j: (
            sum(m.x[i, j] + sum(m.y[i, j, k] for k in m.K) for i in m.I) + m.f[j]
            == DEMAND[j]
        ),
    )
    m.quality = pyo.Constraint(
        m.J,
        rule=lambda m, j: (
            sum(
                conc[i]
                * (m.x[i, j] + sum((1 - TECH[k][0]) * m.y[i, j, k] for k in m.K))
                for i in m.I
            )
            <= LIMIT[j] * DEMAND[j]
        ),
    )
    m.cap = pyo.Constraint(
        m.K,
        rule=lambda m, k: (
            sum(m.y[i, j, k] for i in m.I for j in m.J) <= TECH[k][1] * m.z[k]
        ),
    )
    m.obj = pyo.Objective(
        expr=sum(m.f[j] for j in m.J)
        + 0.1 * sum(m.d[i] for i in m.I)
        + sum(TECH[k][2] * m.y[i, j, k] for i in m.I for j in m.J for k in m.K)
        + (sum(TECH[k][3] * m.z[k] for k in m.K) if level == 3 else 0)
    )
    m._conc = conc
    return m
```

The common `solve` function calls `appsi_highs`, checks optimal termination before loading a solution, and checks constraint residuals, bounds, and integer domains. After building the model, use:

```
from models.common import solve
m = solve(build(3))
```

18.4 Optimal Solution

The reference objective is **20.8235 USD/h**. This value is optimal for the explicitly stated model and data.

User	Fresh (m3/h)	Direct reuse (m3/h)	Treated reuse (m3/h)	Concentration (mg/L)
U1	5.2941	4.7059	40.0000	30.0000
U2	0.0000	40.0000	0.0000	60.0000

Quantity	Value
Freshwater (m3/h)	5.2941
Disposal (m3/h)	25.2941
Technology	moderate

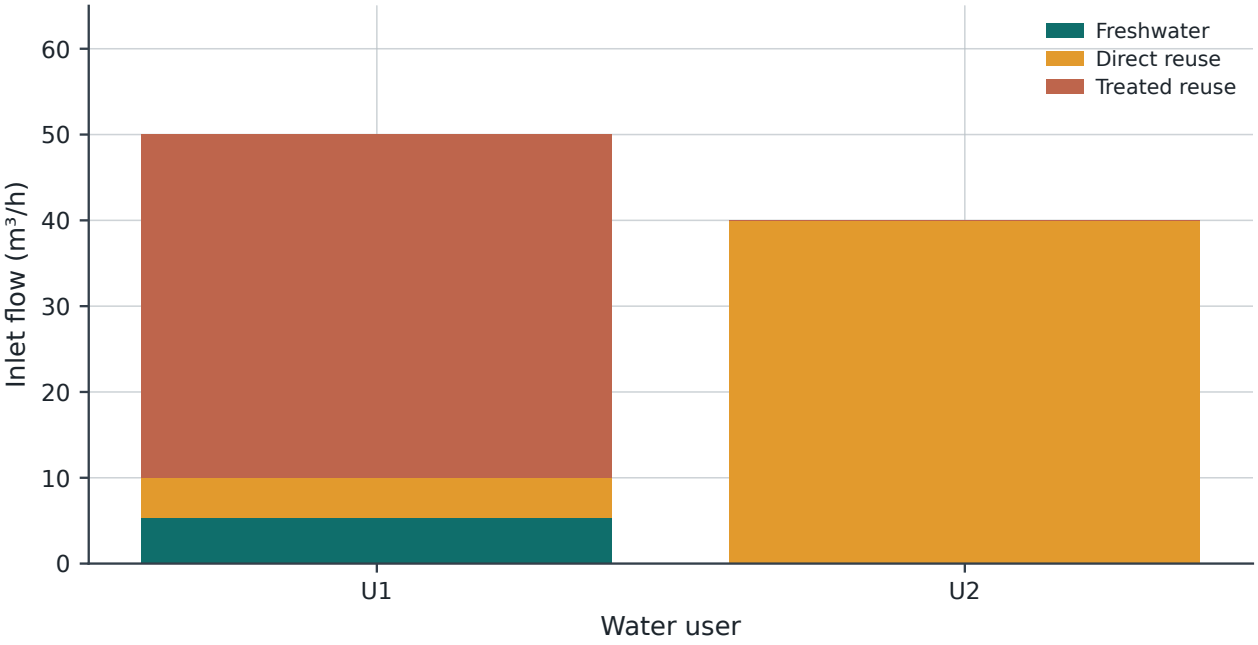

Figure 18.1: Wastewater Treatment and Reuse: reference solution for level 3.

18.4.1 Check your solution

Check the worst-case water and contaminant balances. Fix each of the three choices (none, moderate, advanced), solve its allocation LP, and compare the feasible costs.

The automated residual, bound, and integrality audit passed with a maximum violation of 0.00e+00 in model units. Domain-specific checks passed. Model units differ across equations, so this numerical audit does not replace dimensional analysis.

18.5 Brief Discussion

The solution uses 5.294 m3/h of freshwater and discharges 25.294 m3/h, at 20.824 USD/h. Treatment choice: moderate. Contaminant removed by treatment must leave in the captured-residue stream. Fixed removal and segregated paths are substantive process assumptions, not just numerical conveniences.

18.5.1 Extension to investigate

Propose an unrestricted pool formulation with flow and concentration variables. Identify its bilinear equations and explain what additional measurements or bounds would be required before solving.

Sources and Further Reading

All process data, kinetic coefficients, costs, uncertainty bounds, and operating scenarios in this volume are synthetic teaching assumptions. They are fully specified in the problem statements and Python modules. The examples are original instructional instances; no industrial performance or safety guarantee is implied.

Software references:

- Pyomo, **APPSI HiGHS interface**. Official solver documentation. Used for the linear and mixed-integer reference models.
- Quarto, **Book Structure**. Official book documentation. Used to organize the six parts and separate editions.
- Quarto, **PDF Basics**. Official PDF documentation. Used for the A4 XeLaTeX publication output.

The robust blending counterpart follows directly from the LP dual of the stated uncertainty support function. The scheduling check follows exhaustive enumeration of four-job permutations. The reactor formulas follow the stated first-order, constant-density, isothermal assumptions. Students should derive these relationships from the balances and assumptions shown in the relevant chapter rather than treat them as universal process laws.