



Ahsanullah University of Science and Technology (AUST)
Department of Mechanical and Production Engineering

LABORATORY MANUAL

For the students of

Department of Mechanical and Production Engineering
3rd Year, 2nd Semester

Student Name :
Student ID :

Department of Mechanical and Production Engineering
Ahsanullah University of Science and Technology (AUST)

IPE 3210: Optimization Modelling and Methods Sessional

Credit Hour: 1.5

List of Experiments:

1. Introduction to Optimization Techniques (Classical and Advanced) and Formulating Optimization Models and Algebraic Representation Using LINDO Software.
2. Design and implementation of Service and Queuing system
3. Formulate and Solve Scheduling Problems (Floor Shop, Job Shop, Mixed Shop Scheduling) Using LEKIN
4. Intermediate Optimization Modelling — Single-Item Capacitated Lot Sizing Problem
5. Formulate and solve Network flow model
6. Intermediate optimization modelling application (performance - assessment problem)
7. Intermediate Optimization Modelling Application (Inventory Control Problem)
8. Advanced optimization modelling application (Facility Location Problem)
9. Multi criteria and Goal programming techniques.

Total Marks		
Assignment/Project	Attendance	Quiz
40	10	50

Experiment 1: Introduction to Optimization Techniques (Classical and Advanced) and Formulating Optimization Models and Algebraic Representation Using LINDO Software.

1. Objectives

The objective of this experiment is to introduce students to the basic and applied concepts of optimization modeling and to develop their ability to solve an industrial production planning problem using LINDO software.

- Understand the meaning, scope, and importance of optimization techniques in engineering and industrial decision-making.
- Differentiate between classical optimization techniques and advanced optimization techniques.
- Identify decision variables, objective functions, constraints, parameters, and non-negativity requirements in an optimization model.
- Formulate a practical industrial production planning problem mathematically.
- Represent optimization problems in clear algebraic form using standard notation.
- Develop an executable optimization model using LINDO software syntax.
- Solve the optimization model and interpret the objective value, variable values, slack/surplus, binding constraints, reduced costs, and sensitivity information.
- Relate the mathematical solution to practical production planning decisions in an industrial environment.

2. Introduction

Optimization is the scientific process of selecting the best possible decision from a set of feasible alternatives. In engineering and manufacturing systems, decision makers frequently face limitations related to machine capacity, labor availability, raw materials, budget, delivery time, energy consumption, quality requirements, and customer demand. Optimization provides a systematic mathematical approach for using these limited resources effectively while achieving a desired objective such as maximizing profit, minimizing cost, reducing production time, improving productivity, or increasing service level.

In Industrial and Production Engineering, optimization is one of the most important analytical tools because industrial systems are usually complex and resource constrained. A production manager may need to decide how many units of each product should be manufactured, which machine should process which job, how much inventory should be maintained, where a facility should be located, or how workers should be assigned to different tasks. Without optimization, such decisions may be made by intuition or trial and error, which can lead to inefficient resource utilization, unnecessary cost, and poor operational performance. By contrast, optimization transforms the real problem into a structured mathematical model and provides a logical basis for decision-making.

Optimization problems can be classified in several ways depending on the nature of the objective function, constraints, and decision variables. If the objective function and all constraints are linear, the problem is known as a Linear Programming problem. If some or all variables must take whole-number values, it becomes an Integer Programming or Mixed Integer Programming problem. If the model contains nonlinear equations, it is classified as a Nonlinear Programming problem. Some problems involve sequential decision-making over time and are handled by Dynamic Programming. In modern engineering applications, many problems also involve multiple conflicting objectives, such as minimizing cost while maximizing quality, which leads to Multi-objective Optimization.

Classical optimization techniques are based on mathematical programming and analytical procedures. These include Linear Programming, Integer Programming, Nonlinear Programming, Dynamic Programming, and calculus-based methods such as gradient search and Lagrange multipliers. Classical methods are powerful when the problem structure is well-defined and the mathematical model can be formulated accurately. They are widely used in production planning, blending problems, transportation planning, workforce scheduling, aggregate planning, quality control, and resource allocation.

Advanced optimization techniques are generally used for complex, nonlinear, discontinuous, or large-scale problems where exact mathematical methods may become difficult or computationally expensive. These methods include Genetic Algorithm, Particle Swarm Optimization, Ant Colony Optimization, Simulated Annealing, Tabu Search, Differential Evolution, and other metaheuristic approaches. Many of these algorithms are inspired by natural, biological, or physical phenomena and are designed to search a large solution space effectively. Although they may not always guarantee a mathematically exact global optimum, they are highly useful for engineering design optimization, machine scheduling, supply chain network design, parameter tuning, and complex manufacturing problems.

Real-life industrial applications of optimization are extensive. In manufacturing, optimization is used to determine optimal production quantities, minimize waste, reduce setup time, schedule jobs on machines, and allocate workers to operations. In supply chain systems, it supports transportation planning, warehouse location, distribution routing, and inventory control. In quality engineering, optimization helps identify process parameters that improve product quality and reduce defects. In energy systems, it is applied to minimize power generation cost and optimize renewable energy integration. In product design, it can be used to select materials, minimize weight, and improve mechanical performance.

The foundation of any optimization study is model formulation. A mathematical optimization model normally consists of decision variables, an objective function, constraints, and parameters. Decision variables represent unknown quantities to be determined. The objective function measures the performance of the system. Constraints represent restrictions that must be satisfied. Parameters are known numerical values such as profit per unit, processing time, machine capacity, labor availability, and material consumption. A well-formulated model is essential because the quality of the optimization result depends heavily on the correctness of the model.

LINDO is a widely used optimization software package designed for solving linear, integer, and nonlinear optimization models. It allows students and practitioners to write optimization problems in a

simple algebraic format and obtain results such as optimal decision variable values, objective function value, slack or surplus, reduced costs, dual prices, and sensitivity ranges. Therefore, LINDO is highly suitable for introducing undergraduate engineering students to optimization modeling, solution procedures, and interpretation of results. In this experiment, students will formulate a multi-product production planning model and solve it using LINDO.

3. Theoretical Background

3.1 Decision Variables

Decision variables are the unknown quantities whose values must be determined by solving the optimization model. They represent controllable decisions in the system. In a production planning problem, decision variables may indicate the number of units or production lots of different products.

$X_j = \text{quantity or production lot of product } j, j = 1, 2, \dots, n$

3.2 Objective Function

The objective function mathematically expresses the goal of the optimization problem. It may be a maximization objective, such as profit or productivity, or a minimization objective, such as cost, waste, energy consumption, or processing time.

Maximize $Z = c_1X_1 + c_2X_2 + \dots + c_nX_n$ Minimize $Z = c_1X_1 + c_2X_2 + \dots + c_nX_n$

3.3 Constraints

Constraints are mathematical restrictions that define the limits within which the decision variables must operate. They may represent resource capacity, market demand, production balance, quality requirements, budget limits, or technological conditions.

$a_{11}X_1 + a_{12}X_2 + \dots + a_{1n}X_n \leq b_1$
 $a_{21}X_1 + a_{22}X_2 + \dots + a_{2n}X_n \leq b_2$

3.4 Feasible Region

The feasible region is the set of all decision variable combinations that satisfy every constraint in the model. Any solution outside this region is infeasible because it violates at least one restriction. In a two-variable linear programming problem, the feasible region can be represented graphically as a polygon bounded by the constraint lines.

3.5 Optimal Solution

An optimal solution is the feasible solution that gives the best value of the objective function. For maximization problems, it gives the highest possible objective value, while for minimization problems, it gives the lowest possible objective value. A model may have a unique optimum, multiple optima, no feasible solution, or an unbounded solution depending on its structure.

3.6 Linear Programming

Linear Programming is an optimization technique in which the objective function and all constraints are linear functions of the decision variables. It is extensively used for production planning, blending, transportation, assignment, and resource allocation problems.

Optimize $Z = \sum c_j X_j$ *Subject to* $\sum a_{ij} X_j \leq b_i, X_j \geq 0$

3.7 Integer Programming

Integer Programming is used when some or all decision variables must take integer values. This is important when fractional values are not practical, such as the number of machines, workers, vehicles, product batches, or facility locations.

$X_j = 0, 1, 2, 3, \dots$

A special case is Binary Integer Programming, where variables can take only 0 or 1 values to represent yes/no decisions.

$$Y_j \in \{0, 1\}$$

3.8 Nonlinear Programming

Nonlinear Programming involves at least one nonlinear objective function or nonlinear constraint. These models are common in engineering design, process optimization, structural optimization, and material parameter optimization. Nonlinear models may have local optima, making them more difficult to solve than linear models.

Optimize $Z = f(X_1, X_2, \dots, X_n)$

3.9 Dynamic Programming

Dynamic Programming is a method for solving multi-stage decision problems by dividing a complex problem into smaller subproblems. It is useful in inventory control, equipment replacement, shortest path problems, reliability analysis, and production scheduling over multiple time periods.

3.10 Multi-objective Optimization

Multi-objective optimization deals with problems that have two or more objectives to optimize simultaneously. In many industrial systems, objectives conflict with each other. For example, reducing cost may reduce quality, while increasing quality may require more processing time. The outcome is usually represented by a set of trade-off solutions known as Pareto optimal solutions.

Minimize/Maximize $F(X) = [f_1(X), f_2(X), \dots, f_k(X)]$

3.11 Metaheuristic Optimization

Metaheuristic optimization methods are advanced search techniques that are useful for complex and large-scale problems. Examples include Genetic Algorithm, Particle Swarm Optimization, Ant Colony Optimization, Simulated Annealing, Tabu Search, and Differential Evolution. These techniques are commonly used when traditional exact methods are difficult to apply because of nonlinearity, discontinuity, large search space, or combinatorial complexity.

4. Introduction to LINDO Software

4.1 What is LINDO?

LINDO is an optimization software package used to formulate and solve mathematical programming models. It is particularly suitable for Linear Programming, Integer Programming, and Nonlinear Programming models. The software uses a simple algebraic syntax, making it convenient for students to translate a mathematical optimization model directly into a computer-solvable format.

4.2 Features of LINDO

- Simple algebraic model input format.
- Capability to solve linear, integer, and nonlinear optimization problems.
- Provides optimal objective value and decision variable values.
- Generates slack or surplus values for constraints.
- Reports reduced costs and dual prices for sensitivity interpretation.
- Supports sensitivity analysis for objective coefficients and right-hand-side values.
- Useful for production planning, transportation, blending, scheduling, and resource allocation problems.

4.3 Advantages of LINDO

- Easy for beginners because the model is written close to standard mathematical notation.
- Suitable for classroom teaching and laboratory exercises.
- Fast solution for small and medium-sized optimization problems.
- Helps students connect algebraic formulation with computational results.
- Provides useful post-optimality information for managerial interpretation.

4.4 Industrial Applications

- Production planning and product mix optimization.
- Machine capacity planning and bottleneck analysis.
- Workforce allocation and shift scheduling.
- Transportation and distribution planning.
- Inventory and warehouse planning.
- Material blending and process industry optimization.

4.5 Installation Overview

The installation process depends on the version of LINDO provided by the institution. Generally, the user downloads or obtains the installer, runs the setup file, accepts the license agreement, selects the installation location, and completes the installation wizard. After installation, the software can be opened from the desktop shortcut or the Start menu. Students should use the licensed version available in the laboratory.

4.6 Interface Explanation and Model Solving Steps

Table 1: Major components of the LINDO interface

Interface Component	Purpose
Model Editor	Area where the mathematical model is typed using LINDO syntax.
Solve Command	Runs the optimization solver and generates the solution.
Solution Report	Displays objective value, variable values, reduced costs, slack/surplus, and dual prices.
Range/Sensitivity Report	Provides allowable changes in coefficients and RHS values where the current basis remains valid.
Error Messages	Indicate syntax or modeling errors that must be corrected before solving.

5. Problem Statement

A manufacturing company produces four industrial products: Product A, Product B, Product C, and Product D. These products are manufactured using limited CNC machining hours, assembly hours, finishing hours, raw material, and skilled labor hours. The company wants to determine the number of production lots for each product that should be manufactured during the next planning period to maximize total profit while satisfying resource limitations and market-related restrictions.

The products are assumed to be produced in production lots. Therefore, a value of $X_1 = 55$ means 55 production lots of Product A. The laboratory model is first solved as a Linear Programming model. Students may later extend the model by forcing the decision variables to be integers.

Table 2: Production planning data for the industrial optimization problem

Item	Product A	Product B	Product C	Product D	Available
Profit per lot (BDT)	640	400	960	365	-
CNC machining hours/lot	4	5	6	2	600
Assembly hours/lot	2	3	2	1	300
Finishing hours/lot	2	1	3	1	260
Raw material kg/lot	7	9	8	3	900
Skilled labor hours/lot	3	4	5	2	470

Table 3: Market and production policy restrictions

Restriction	Description
Product A	Minimum 20 lots and maximum 90 lots
Product B	Minimum 15 lots and maximum 80 lots
Product C	Minimum 10 lots and maximum 70 lots
Product D	Minimum 25 lots and maximum 120 lots
Total production	At least 120 lots must be produced
Product C market mix	Product C must not exceed 35% of total production
Product D market mix	Product D must be at least 20% of total production

5.1 Decision Variables

Variable	Meaning
X_1	Number of production lots of Product A
X_2	Number of production lots of Product B

X3	Number of production lots of Product C
X4	Number of production lots of Product D

6. Algebraic Representation

6.1 Objective Function

The objective is to maximize total profit generated from the production of all four products.

$$\text{Maximize } Z = 640X1 + 400X2 + 960X3 + 365X4$$

6.2 Resource Constraints

$4X1 + 5X2 + 6X3 + 2X4 \leq 600$ (CNC machining hours)
 $2X1 + 3X2 + 2X3 + X4 \leq 300$ (Assembly hours)
 $2X1 + X2 + 3X3 + X4 \leq 260$ (Finishing hours)
 $7X1 + 9X2 + 8X3 + 3X4 \leq 900$ (Raw material)
 $3X1 + 4X2 + 5X3 + 2X4 \leq 470$ (Skilled labor hours)

6.3 Demand and Policy Constraints

$$20 \leq X1 \leq 90$$

$$15 \leq X2 \leq 80$$

$$10 \leq X3 \leq 70$$

$$25 \leq X4 \leq 120$$

$$X1 + X2 + X3 + X4 \geq 120 \text{ (Minimum total production)}$$

$$X3 \leq 0.35(X1 + X2 + X3 + X4)$$

For LINDO, the Product C market mix constraint is written in linear form as:

$$13X3 - 7X1 - 7X2 - 7X4 \leq 0$$

For LINDO, the Product D market mix constraint is written in linear form as:

$$X1 + X2 + X3 - 4X4 \leq 0$$

6.4 Non-negativity Conditions

$$X1, X2, X3, X4 \geq 0$$

7. LINDO Code

The following model is written in a syntax suitable for the classic LINDO interface. Constraint names are placed before the right parenthesis to make the output easier to interpret. Some LINDO versions do not accept inline comments; therefore, the executable model is provided without comments, and the explanatory notes are given after the code.

```

MAX = 640 X1 + 400 X2 + 960 X3 + 365 X4 ST

CNC) 4 X1 + 5 X2 + 6 X3 + 2 X4 <= 600
ASSEMBLY) 2 X1 + 3 X2 + 2 X3 + X4 <= 300
FINISHING) 2 X1 + X2 + 3 X3 + X4 <= 260 MATERIAL)
7 X1 + 9 X2 + 8 X3 + 3 X4 <= 900
LABOR) 3 X1 + 4 X2 + 5 X3 + 2 X4 <= 470

MIN_X1) X1 >= 20 MAX_X1)
X1 <= 90 MIN_X2) X2 >= 15
MAX_X2) X2 <= 80 MIN_X3)
X3 >= 10 MAX_X3) X3 <= 70
MIN_X4) X4 >= 25 MAX_X4)
X4 <= 120

MIN_TOTAL) X1 + X2 + X3 + X4 >= 120
MIX_C) 13 X3 - 7 X1 - 7 X2 - 7 X4 <= 0
MIX_D) X1 + X2 + X3 - 4 X4 <= 0

END

```

Table 4: Explanation of LINDO model sections

Code Section	Explanation
MAX	Defines the objective function for maximizing total profit.
ST	Indicates the beginning of the constraint section.
CNC, ASSEMBLY, FINISHING, MATERIAL, LABOR	Represent the available production resources.
MIN_X and MAX_X constraints	Represent product-specific minimum and maximum production lot requirements.
MIN_TOTAL	Ensures that total production is not below 120 lots.
MIX_C	Ensures that Product C does not exceed 35% of total production.
MIX_D	Ensures that Product D is at least 20% of total production.
END	Indicates the end of the model.

7.1 Optional Integer Programming Extension

If the company requires production lots to be whole numbers, the following line may be added before END. This converts the model from Linear Programming to Integer Programming and introduces an advanced optimization concept.

GIN X1 X2 X3 X4

8. Procedure

1. Open LINDO software from the desktop shortcut or Start menu.
2. Select File > New to open a new model editor window.
3. Type the complete LINDO code exactly as shown in Section 7.
4. Check that the objective function begins with MAX and that the constraint section begins with ST.
5. Ensure that each constraint is written on a separate line and that the model ends with END.
6. Click Solve or choose the Solve command from the menu.
7. If an error message appears, identify the line number and correct spelling, signs, constraint symbols, or missing terms.
8. After the model is solved, record the objective function value and the values of X1, X2, X3, and X4.
9. Record slack or surplus values for each constraint and identify which constraints are binding.
10. Observe the reduced costs and dual prices reported by LINDO.
11. Open or view the sensitivity/range report if available in the installed version.
12. Interpret the optimal production plan in practical industrial terms.

9. Expected Output

When the linear programming model is entered correctly, the expected optimal production plan is shown below. Minor formatting differences may appear depending on the LINDO version used in the laboratory.

Table 5: Expected optimal solution

Symbol	Meaning	Expected Value
X1	Product A	55 lots
X2	Product B	15 lots
X3	Product C	25 lots
X4	Product D	60 lots
Z	Maximum profit	87,100 BDT

Table 6: Resource usage and binding constraint interpretation

Constraint	Used/Actual	Limit	Slack/Surplus	Status
CNC machining	565	600	35	Non-binding
Assembly	265	300	35	Non-binding
Finishing	260	260	0	Binding
Raw material	900	900	0	Binding
Skilled labor	470	470	0	Binding
Minimum total production	155	120 minimum	35 surplus	Non-binding

Product C mix	Product C = 16.13% of total	$\leq 35\%$	Satisfied	Non-binding
Product D mix	Product D = 38.71% of total	$\geq 20\%$	Satisfied	Non-binding

9.1 Interpretation of Objective Function Value

The optimal objective function value is 87,100 BDT. This means that, under the given resource capacities and market restrictions, the company can obtain a maximum profit of 87,100 BDT by producing 55 lots of Product A, 15 lots of Product B, 25 lots of Product C, and 60 lots of Product D.

$$Z = 640(55) + 400(15) + 960(25) + 365(60) = 87,100 \text{ BDT}$$

9.2 Slack, Surplus, and Binding Constraints

A binding constraint has zero slack or zero surplus and directly limits the objective function value. In this model, finishing hours, raw material, and skilled labor are binding constraints. CNC machining and assembly have 35 unused hours each; therefore, increasing CNC or assembly capacity alone will not improve profit unless the binding resources are also increased.

9.3 Reduced Costs

Reduced cost indicates how much the objective coefficient of a non-basic variable must improve before that variable can enter the optimal solution. For variables already produced above their lower bound, the reduced cost is normally zero. If a variable remains at a lower or upper bound, the reduced cost helps explain whether producing more or less of that product would improve the objective value.

9.4 Sensitivity Analysis

Sensitivity analysis examines how changes in profit coefficients or resource availability affect the optimal solution. In this problem, the binding resources are finishing hours, raw material, and skilled labor. Therefore, these resources are expected to have positive economic value within allowable ranges. The non-binding resources, CNC machining and assembly, are expected to have zero shadow price because unused capacity remains after the optimal plan is selected.

10. Discussion

10.1 Interpretation of Results

The optimal solution indicates that Product A, Product C, and Product D should be produced in substantial quantities, while Product B should be produced only at its required minimum level. This suggests that Product B is less attractive under the current profit and resource consumption structure. The result also shows that the company is limited mainly by finishing, raw material, and skilled labor capacities. These resources should be considered bottlenecks in the production system.

10.2 Practical Implications

From a managerial viewpoint, the solution provides a production plan that maximizes profit while satisfying all operational and market restrictions. If management wants to increase profit further, it should investigate whether additional finishing capacity, raw material, or skilled labor can be

obtained. However, increasing CNC machining or assembly capacity alone is not expected to improve profit because these resources have unused capacity in the current solution.

10.3 Common Modeling Mistakes

- Writing the objective function with incorrect signs or missing variables.
- Using incorrect units, such as mixing hours per unit with hours per lot.
- Forgetting non-negativity or minimum production constraints.
- Entering $>$ instead of $\geq$, or $<$ instead of $\leq$, in LINDO syntax.
- Formulating percentage constraints incorrectly, especially product-mix constraints.
- Forgetting to include END at the end of the model.
- Assuming that a non-binding resource will improve profit if increased.
- Interpreting mathematical results without considering practical industrial feasibility.

10.4 Industrial Significance

Optimization allows industries to make rational and evidence-based decisions under limited resources. The same modeling approach used in this laboratory can be applied to real production systems involving multiple products, machines, labor groups, raw materials, and market demand conditions. In modern manufacturing, such models help reduce waste, improve productivity, increase profitability, identify bottlenecks, and support strategic capacity planning.

10.5 Link Between Classical and Advanced Optimization

The present experiment primarily uses Linear Programming, which is a classical optimization technique. However, the model can be extended to advanced optimization by adding integer restrictions, binary setup decisions, nonlinear cost functions, or multiple objectives. For example, if each product must be produced in whole-number lots, the model becomes an Integer Programming problem. If the company wants to maximize profit and minimize environmental impact simultaneously, the model becomes a Multi-objective Optimization problem. If the problem becomes very large or nonlinear, metaheuristic methods such as Genetic Algorithm or Particle Swarm Optimization may be considered.

References for Further Reading

Hillier, F. S., and Lieberman, G. J., Introduction to Operations Research, McGraw-Hill.

Taha, H. A., Operations Research: An Introduction, Pearson.

Winston, W. L., Operations Research: Applications and Algorithms, Cengage Learning.

LINDO Systems Inc., LINDO User Manual and Optimization Modeling Documentation.

Experiment 2: Design and implementation of Service and Queuing system

Objective

The objective of this experiment is to:

- Simulate a single-server checkout system at a grocery store by modeling random customer arrivals and service times using the provided probability distributions.
- Calculate and analyze the seven performance metrics required to evaluate the efficiency of this specific checkout counter, such as average wait time and server idle fraction.
- To validate the efficacy of using MATLAB to automate the discrete-event simulation process as a more accurate and efficient alternative to manual calculation.

Software

- Matlab

Introduction

Queuing theory serves as a fundamental discipline within operations research, providing the mathematical architecture necessary to model and analyze the flow of entities through limited-capacity service channels. By examining the stochastic nature of arrivals and the variability in service durations, analysts can identify system bottlenecks, quantify waiting time patterns, and optimize resource allocation. Traditionally, the analysis of these systems relied on tedious, manual, step-by-step arithmetic—a process prone to human error and limited in scalability. Today, computational platforms like MATLAB have revolutionized this field by enabling the rapid, automated execution of discrete-event simulations. By leveraging MATLAB's capacity for iterative logic and vector-based computation, engineers can transform abstract probability distributions into concrete, data-driven insights. This transition from static analysis to dynamic modeling not only enhances the accuracy of system performance metrics but also provides a robust framework for testing operational strategies in real-world scenarios, making it an essential skill for modern systems engineering.

Beyond the immediate benefits of automation, the use of programming environments to perform these simulations fosters a deeper understanding of system dynamics that manual calculation often obscures. When engineers build simulations from the ground up, they must explicitly define the state transitions—such as arrival, service entry, and service completion—which forces a granular view of how individual entities impact the system's overall stability.

Problem

A small grocery store has only one checkout counter. Customers arrive at this checkout counter at random from 1 to 8 minutes apart. Each possible value of interarrival time has the same probability of occurrence. The service times vary from 1 to 6 minutes with the probabilities shown in Table 1. The problem is to analyse the system by simulating the arrival and service of 6 customers. Using the sequence of random numbers in Table 2 determine the following:

1. The average waiting time for a customer
2. The probability that a customer has to wait in the queue
3. The fraction of idle time of the server
4. The average service time
5. The average time between arrivals
6. The average waiting time of those who wait
7. The average time a customer spends in the system

Table 1: Probability of Service Times

Service Time (Mins)	Probability
1	0.1
2	0.2
3	0.3
4	0.25
5	0.1
6	0.05

Table 2: Random Digits for Arrival and Service Time

Random digits for Arrival		913	727	15	948	309
Random digits for service time	84	10	74	53	17	79

Code

```
% Queue Simulation Script

n = 6;

% Random digits from Table 2

% Note: Using 0 for Cust 1 as it is the first arrival

arr_rand = [0, 913, 727, 15, 948, 309];

serv_rand = [84, 10, 74, 53, 17, 79];


% 1. Map arrivals (1-8 min) - Uniform 1/8 chance

% Mapping: 000-124:1, 125-249:2, ..., 875-999:8

inter_arr = [0, floor(arr_rand(2:end)/125) + 1];


% 2. Map service times based on Table 1 probabilities

serv_times = zeros(1, n);

for i = 1:n

    r = serv_rand(i);

    if r < 10, serv_times(i) = 1;

    elseif r < 30, serv_times(i) = 2;

    elseif r < 60, serv_times(i) = 3;

    elseif r < 85, serv_times(i) = 4;

    elseif r < 95, serv_times(i) = 5;

    else, serv_times(i) = 6;

end

end
```

% 3. Calculate Timings

```
arrival_time = cumsum(inter_arr);
```

```
service_start = zeros(1, n);
```

```
wait_time = zeros(1, n);
```

```
service_end = zeros(1, n);
```

```
service_start(1) = arrival_time(1);
```

```
service_end(1) = service_start(1) + serv_times(1);
```

```
for i = 2:n
```

```
    service_start(i) = max(arrival_time(i), service_end(i-1));
```

```
    wait_time(i) = service_start(i) - arrival_time(i);
```

```
    service_end(i) = service_start(i) + serv_times(i);
```

```
end
```

```
system_time = service_end - arrival_time;
```

% 4. Display Results

```
fprintf('--- Simulation Results ---\n');
```

```
fprintf('1. Avg Waiting Time: %.2f\n', mean(wait_time));
```

```
fprintf('2. Prob. of Waiting: %.2f\n', sum(wait_time > 0) / n);
```

```
fprintf('3. Fraction of Idle Time: %.2f\n', 1 - (sum(serv_times) / service_end(end)));
```

```
fprintf('4. Avg Service Time: %.2f\n', mean(serv_times));
```

```
fprintf('5. Avg Time Between Arrivals: %.2f\n', mean(inter_arr(2:end)));
```

```
fprintf('6. Avg Wait Time (of those who wait): %.2f\n', mean(wait_time(wait_time > 0)));
```

```
fprintf('7. Avg Time in System: %.2f\n', mean(system_time));
```

```
% Display table for verification
```

```
ResultsTable = table((1:n)', arrival_time', serv_times', service_start', wait_time', service_end',  
system_time', ...
```

```
    'VariableNames', {'Cust', 'Arrival', 'ServiceTime', 'Start', 'Wait', 'End', 'SystemTime'});
```

```
disp(ResultsTable);
```

Assignment

Solve the same problem using the following modified values:

Table 1: Probability of Service Times

Service Time (Mins)	Probability
1	0.15
2	0.25
3	0.25
4	0.20
5	0.10
6	0.05

Table 2: Random Digits for Arrival and Service Time

Random digits for Arrival		412	805	221	674	159
Random digits for service time	22	78	05	55	91	34

Experiment 3: Formulate and Solve Scheduling Problems (Floor Shop, Job Shop, Mixed Shop Scheduling) Using LEKIN

Objectives

- To formulate and model single-machine and flow-shop manufacturing scheduling problems from raw production data under distinct constraints like release dates (r_j), due dates (d_j), and job priority weights (w_j).
- To configure and execute problem instances within the LEKIN Scheduling System workspace, correctly mapping real-world operational parameters into the software environment.
- To evaluate and compare the performance of different scheduling dispatching rules (such as EDD, FCFS, and SPT) in optimizing specific shop floor metrics like Makespan ($C_{\max}$) and Total Weighted Tardiness $\sum w_j T_j$.
- To interpret visual Gantt charts generated by LEKIN to verify schedule feasibility, identify machine idle times, and analyze the impact of priority weights on job sequencing.

Software Used

- LEKIN Scheduling System

Introduction

2. Introduction & Theoretical Background

Production scheduling involves allocating scarce machine resources over time to optimize manufacturing efficiency. Depending on shop floor layout, environments range from **Single-Machine** systems (one shared asset) and **Flow Shops** (jobs following an identical, linear machine sequence) to **Job Shops** and **Mixed Shops** (jobs following customized, variable routings).

To evaluate schedule efficiency, operations rely on two primary metrics:

- **Makespan ($C_{\max}$):** The total time required to complete all jobs. Minimizing makespan maximizes machine utilization. In environments with **Release Dates (r_j)**—the earliest a job can start—schedulers aim for a **Non-Delay Schedule**, meaning no machine sits idle if a job is waiting in the queue.
- **Total Weighted Tardiness $\sum w_j T_j$:** A customer-focused metric penalizing jobs that finish past their **Due Date (d_j)**. **Tardiness (T_j)** is calculated as $\max(0, C_j - d_j)$.

Incorporating a priority Weight (w_j) ensures high-value jobs are prioritized to avoid steep late penalties.

The **LEKIN Scheduling Software** integrates these constraints and objectives, utilizing heuristic dispatching rules to generate feasible sequences and visualize production timelines through interactive Gantt charts.

Problem Statement

Non Delay Schedule

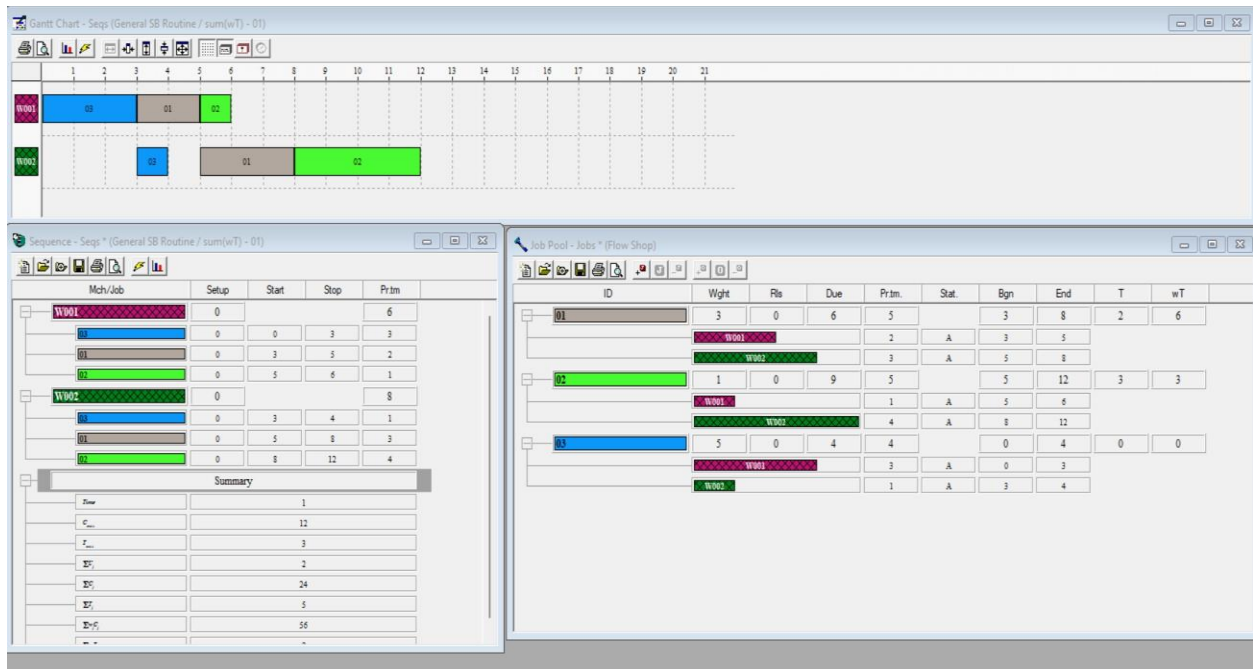
1. C_{max}

- given: n jobs with processing times $p_1, \dots, p_n$ and release dates $r_1, \dots, r_n$
- jobs have to be scheduled without preemption on one machine taking into account the earliest starting times of the jobs, such that the makespan is minimized
- $n=4$, $p=(2,4,2,3)$, $r=(5,4,0,3)$
- Feasible Schedule with $C_{max} = 12$

2. $\sum w_j T_j$

- given: n jobs with weights $w_1, \dots, w_n$ and due dates $d_1, \dots, d_n$
- Operations (i,j) with processing times p_{ij} , $i = 1,2$; $j = 1, \dots, n$
- Jobs have to be scheduled on two machines such that operation $(2,j)$ is scheduled on machine 2 and does not start before operation $(1,j)$, which is scheduled on machine 1, is finished and the weighted tardiness is minimized
- $n=3$, $p = \begin{pmatrix} 2 & 1 & 3 \\ 3 & 4 & 1 \end{pmatrix}$, $w = (3,1,5)$, $d = (6,9,4)$

LEKIN Scheduling (2)



$\sum w_i$	56
$\sum f_i$	9

Assignment

Solve Problem Statement -1

Experiment 4: Intermediate Optimization Modelling — Single-Item Capacitated Lot Sizing Problem

Objectives

- To analyze how production capacity ceilings alter inventory cost-minimization strategies.
- To formulate a Single-Item Capacitated Lot Sizing problem using Mixed-Integer Linear Programming (MILP).
- To implement a single-dimensional time-indexed model within IBM ILOG CPLEX Optimization Studio.
- To observe how optimization algorithms shift production backward in time to proactively resolve asset or capacity bottlenecks.

Software Used

- **IBM ILOG CPLEX Optimization Studio** (OPL Modelling Language and CPLEX Solver engine).

Introduction

In standard textbook lot-sizing models, it is assumed that a factory has infinite capacity and can produce any quantity of goods instantly. In real-world industrial environments, however, manufacturing lines are bounded by sharp physical bottlenecks, such as limited machine hours, labor shifts, or safety capacities.

When a strict Production Capacity Limit is introduced, a company can no longer wait until a high-demand week to manufacture everything just-in-time. Instead, the optimization engine must dynamically evaluate whether it is more cost-effective to pay a Holding Cost to manufacture products early and store them, or pay extra Setup Costs by running machine changeovers frequently.

Problem Statement

An assembly plant manufactures high-end electric scooters. The operations manager needs to establish a minimum-cost manufacturing plan over a 4-week planning horizon.

The operational parameters are defined as follows:

- **Planning Horizon:** Periods = {1, 2, 3, 4}
- **Weekly Customer Demand:** Week 1 = 20, Week 2 = 30, Week 3 = 80, Week 4 = 10
- **Setup Cost (S):** \$1,500 flat fee every time the production line is activated.
- **Holding Cost (H):** \$5 per scooter per week left in inventory at the end of a period.

- **Production Capacity Ceiling:** The assembly line can produce a maximum of 50 scooters per week.
- **Initial Inventory:** The plant begins Week 1 with 0 scooters.

Mathematical Model Formulation

Objective Function:

$$\text{Minimize } Z = \sum_{t=1}^4 (\text{SetupCost} \cdot y[t] + \text{HoldingCost} \cdot \text{inventory}[t])$$

Subject to Constraints:

1. **Inventory Balance Equation for the First Week:**
 $x[1] - \text{Demand}[1] = \text{inventory}[1]$
2. **Inventory Balance Equations for Subsequent Weeks:**
 $\text{inventory}[t-1] + x[t] - \text{Demand}[t] = \text{inventory}[t] \quad \text{for all } t \in \{2, 3, 4\}$
3. **Hard Production Capacity and Setup Linking Constraint:**
 $x[t] \leq 50 \cdot y[t] \quad \text{for all } t \in \text{Periods}$
4. **Variable Domain Boundaries and Integrality Restrictions:**
 $x[t] \geq 0, \text{ inventory}[t] \geq 0 \text{ and integer} \quad \text{for all } t \in \text{Periods}$
 $y[t] \in \{0, 1\} \quad \text{for all } t \in \text{Periods}$

CPLEX C

```
3  *****/
4  // 1. Parameters & Horizons
5  int T = 4;
6  range Periods = 1..T;
7  int Demand[Periods] = [20, 30, 80, 10];
8  float SetupCost = 1500;
9  float HoldingCost = 5;
10 float MaxCapacity = 50; // Maximum production units per week
11
12 // 2. Decision Variables
13 dvar int+ x[Periods]; // Production quantity
14 dvar int+ inventory[Periods]; // Ending inventory levels
15 dvar boolean y[Periods]; // Binary line setup flag
16 // 3. Objective Function
17 minimize
18   sum(t in Periods) (SetupCost * y[t] + HoldingCost * inventory[t]);
19
20 // 4. Constraints
21 subject to {
22   // Inventory Balance: Week 1
23   x[1] - Demand[1] == inventory[1];
24   // Inventory Balance: Weeks 2 to 4
25   forall(t in 2..T) {
26     inventory[t-1] + x[t] - Demand[t] == inventory[t];
27   }
28   // Hard Production Capacity Limit & Setup Linking Combined
29   // (Since max production is 50, MaxCapacity acts as our Big-M)
30   forall(t in Periods) {
31     x[t] <= MaxCapacity * y[t];
32   }
33 }
34 // 5. Post-Processing Output
35 execute {
36   writeln("--- OPTIMAL CAPACITATED PRODUCTION PLAN ---");
37   for(var t in Periods) {
38     writeln("Week ", t, ": Produced = ", x[t], " | Ending Stock = ", inventory[t], " | Machine Active = ", y[t]);
39   }
40 }
```

Result

Problems Scripting log **Solutions** Conflicts Relaxations Engine log Statistics Profiler

```
// solution (optimal) with objective 4850
// Quality Incumbent solution:
// MILP objective                                4.8500000000e+03
// MILP solution norm |x| (Total, Max)           2.13000e+02  5.00000e+01
// MILP solution error (Ax=b) (Total, Max)       0.00000e+00  0.00000e+00
// MILP x bound error (Total, Max)              0.00000e+00  0.00000e+00
// MILP x integrality error (Total, Max)         0.00000e+00  0.00000e+00
// MILP slack bound error (Total, Max)          0.00000e+00  0.00000e+00
//
```

```
y = [1
      1 1 0];
inventory = [20 40 10 0];
x = [40 50 50 0];
```

Problems **Scripting log** Solutions Conflicts Relaxations Engine log Statistics Profiler

Interactive scripting « Scripting log (drop script code here to execute it)

```
// solution (optimal) with objective 4850
--- OPTIMAL CAPACITATED PRODUCTION PLAN ---
Week 1: Produced = 40 | Ending Stock = 20 | Machine Active = 1
Week 2: Produced = 50 | Ending Stock = 40 | Machine Active = 1
Week 3: Produced = 50 | Ending Stock = 10 | Machine Active = 1
Week 4: Produced = 0 | Ending Stock = 0 | Machine Active = 0
```

Assignment

A sudden spike in overseas container shipping rates causes the fixed setup cost to jump from \$1,500 to \$3,000. Concurrently, warehouse refrigeration and insurance fees drop, lowering the holding cost from \$5 to \$2 per scooter per week. The demand vector and production capacity remain completely unchanged.

- Compare the new setup vector y with your original run. Did the solver turn the machine on more often or less often? Explain the financial logic behind why the optimization engine shifted its strategy.
- Calculate the factory's capacity utilization ($\text{Production} / \text{MaxCapacity}$) for Week 1 under these new settings.

Experiment 5: Formulate and solve Network flow model

Objective

In this lab you will learn how to formulate the Critical Path Method (CPM) of a project network as a linear programming (LP) problem, and how to solve it using the Gurobi optimizer in Python. By the end of this lab you should be able to:

- Model a project network as a directed graph of activities (arcs) and events (nodes).
- Express the longest-path / critical-path problem as a network flow LP.
- Build and solve the LP model using the gurobipy API.
- Interpret the Gurobi output to identify the critical path and project duration.

Software Used:

- Python
- Gurobi
- Tora

Background:

The Critical Path Method determines the minimum time required to complete a project made up of inter-dependent activities. Each activity has a known duration, and activities are linked through precedence relationships represented as a network.

A classical way to find the critical path is to treat it as a longest-path problem: among all paths from the project start node to the project finish node, the critical path is the one with the maximum total duration.

This longest-path problem can be modeled as a network LP by sending one unit of flow from the start node to the finish node and maximizing the total “distance” (duration) traveled. The flow is forced to obey node balance (conservation of flow) at every node:

- At the start node: $\text{outflow} - \text{inflow} = +1$ (one unit of flow originates here).
- At the finish node: $\text{outflow} - \text{inflow} = -1$ (one unit of flow is absorbed here).
- At every intermediate node: $\text{outflow} - \text{inflow} = 0$ (flow simply passes through).

Because the objective pushes flow toward the most rewarding (longest-duration) path, the optimal flow pattern traces out exactly the critical path, and the optimal objective value equals the project duration.

Problem Statement

Arc ($i \rightarrow j$)	Activity	Duration
$1 \rightarrow 2$	A	5
$1 \rightarrow 3$	B	6
$2 \rightarrow 3$	C	3
$2 \rightarrow 4$	D	8
$3 \rightarrow 5$	E	2
$3 \rightarrow 6$	F	11
$4 \rightarrow 5$	Dummy	0
$4 \rightarrow 6$	G	1
$5 \rightarrow 6$	H	12

Mathematical Formulation

Let $x_{ij} \geq 0$ denote the flow on arc (i, j) , and let d_{ij} be the duration of the activity on arc (i, j) . The LP model is:

Maximize $z = \sum d_{ij} \cdot x_{ij}$ (sum over all arcs (i, j))

Subject to, for every node n :

$\sum x_{nj} - \sum x_{in} = b_n$ where $b_n = +1$ if n is the start node, -1 if n is the finish node, 0 otherwise

All decision and deviational variables are non-negative.

Procedure

1. Open a new Python file named `cpm_lp_gurobi.py`.
2. Import `gurobipy` and the `GRB constants` module.
3. Define a dictionary 'arcs' where each key is a (tail, head) node pair and each value is a tuple of (activity name, duration), as shown in Section 4.
4. Identify the `start_node` and `finish_node`, and build a sorted list of all nodes in the network.
5. Create a Gurobi model object and add one continuous, non-negative flow variable $x[i,j]$ for every arc.
6. Set the objective function to maximize the sum of (duration $\times$ flow) over all arcs.
7. For every node, add a flow-balance constraint: outflow minus inflow equals +1 at the start node, -1 at the finish node, and 0 at every other node.
8. Call `model.optimize()` to solve the LP.
9. If the model status is `GRB.OPTIMAL`, print the objective value (project duration) and list every arc whose flow variable $x[i,j]$ is approximately 1 — these arcs form the critical path.
10. Reconstruct the critical path in order from the start node to the finish node and print the activity sequence and the node sequence.

Reference Implementation (Python + PuLP)

```
''''''
```

CPM as a Network LP (Taha, Example 6.5-5)

Longest path from start node (1) to finish node (6) = project duration / critical path.

Idea: treat each activity as an arc (i,j) with "flow" x_{ij} .

Send 1 unit of flow from node 1 to node 6 along the network.

Maximize total duration of the path the flow takes -> longest path = critical path.

Node balance: outflow - inflow = +1 at start node, -1 at finish node, 0 elsewhere.

"""

```
import gurobipy as gp
```

```
from gurobipy import GRB
```

```
# ---- Arcs: (tail, head): (activity name, duration) ----
```

```
arcs = {
```

```
    (1, 2): ("A", 5),
```

```
    (1, 3): ("B", 6),
```

```
    (2, 3): ("C", 3),
```

```
    (2, 4): ("D", 8),
```

```
    (3, 5): ("E", 2),
```

```
    (3, 6): ("F", 11),
```

```
    (4, 5): ("Dummy", 0),
```

```
    (4, 6): ("G", 1),
```

```
    (5, 6): ("H", 12),
```

```
}
```

```
start_node = 1
```

```
finish_node = 6
```

```
nodes = sorted(set(i for i, j in arcs) | set(j for i, j in arcs))
```

```
# ---- Model ----
```

```
model = gp.Model("CPM_as_LP")
```

```
# Flow variables per arc (continuous is fine; LP relaxation is naturally 0/1 here)
```

```
x = model.addVars(arcs.keys(), lb=0, name="x")
```

```
# ---- Objective: maximize total path duration ----
```

```
model.setObjective(  
    gp.quicksum(arcs[(i, j)][1] * x[i, j] for (i, j) in arcs),  
    GRB.MAXIMIZE,  
)
```

```
# ---- Node balance constraints ----
```

```
for n in nodes:
```

```
    outflow = gp.quicksum(x[i, j] for (i, j) in arcs if i == n)
```

```
    inflow = gp.quicksum(x[i, j] for (i, j) in arcs if j == n)
```

```
    if n == start_node:
```

```
        rhs = 1
```

```
    elif n == finish_node:
```

```
        rhs = -1
```

```
    else:
```

```
        rhs = 0
```

```
    model.addConstr(outflow - inflow == rhs, name=f"Node_{n}")
```

```
model.optimize()
```

```
# ---- Report ----
```

```

if model.status == GRB.OPTIMAL:

    print(f"\nProject duration (z) = {model.ObjVal:.0f}\n")

    print("Critical path activities (x_ij = 1):")

    path_arcs = []

    for (i, j), var in x.items():

        if var.X > 0.5:

            name, dur = arcs[(i, j)]

            print(f" {name} (node {i} -> node {j}), duration = {dur}")

            path_arcs.append((i, j, name))

    # Order the path from start to finish for a clean printout

    seq = [start_node]

    current = start_node

    names = []

    while current != finish_node:

        for (i, j, name) in path_arcs:

            if i == current:

                names.append(name)

                current = j

                seq.append(current)

                break

    print("\nCritical path:", " -> ".join(names))

    print("Node sequence:", " -> ".join(str(n) for n in seq))

else:

    print("No optimal solution found. Status code:", model.status)

```

Assignment:

1. Optimize this network flow in PERT method.

Activity	$i-j$	(a, m, b)	Activity	$i-j$	(a, m, b)
<i>A</i>	1-2	(3, 5, 7)	<i>E</i>	3-5	(1, 2, 3)
<i>B</i>	1-3	(4, 6, 8)	<i>F</i>	3-6	(9, 11, 13)
<i>C</i>	2-3	(1, 3, 5)	<i>G</i>	4-6	(1, 1, 1)
<i>D</i>	2-4	(5, 8, 11)	<i>H</i>	5-6	(10, 12, 14)

Figure 1

2. Determine the critical path for the project network in Figure 2

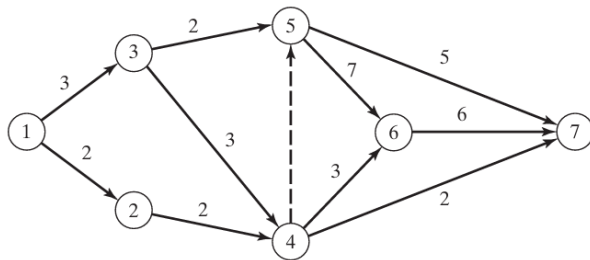


Figure 2

Experiment 6: Intermediate optimization modelling application (performance - assessment problem)

Objectives

- Understand the core concepts of Performance Assessment and benchmarking in operational management.
- Formulate a fractional efficiency optimization problem into a linear programming (LP) model.
- Implement an all-units parallel evaluation model in IBM ILOG CPLEX Optimization Studio.
- Analyze solver outputs to identify high-performing benchmarks versus inefficient operating units.

Software Used

- **IBM ILOG CPLEX Optimization Studio** (OPL Modelling Language and CPLEX Solver engine)

Introduction

In operational management, assessing the performance of business units—such as retail stores, warehouses, or branches—is critical for driving improvements. These units are formally called **Decision-Making Units (DMUs)**. A standard KPI ratio (like Output/Input) works well when there is only one metric to track, but it becomes challenging when units manage a complex mix of multiple resource inputs and performance outputs.

To solve this, we use an optimization technique called **Data Envelopment Analysis (DEA)**. Instead of assigning arbitrary, static weights to performance metrics, DEA allows each individual DMU to choose its own optimal set of weights. The only rule is that these weights must be mathematically fair: when applied to any other unit in the dataset, no unit's efficiency score can exceed 1.0 (100%).

Units that achieve a score of 1.0 form the efficiency frontier (the best-practice benchmarks). Units scoring below 1.0 are inefficient, and their scores indicate how much they could improve relative to the top performers.

Problem Statement

You are managing a small regional chain of three boutique coffee shops (Shop 1, Shop 2, and Shop 3). To plan future training and resource allocations, you need to measure their relative operational efficiency.

To keep the initial analysis clean, each shop is evaluated on exactly one operational resource input and one performance output:

- Input (x): Daily Staffing Hours (The resource consumed).
- Output (y): Daily Revenue in hundreds of dollars (The value generated).

Operational Dataset

Coffee Shop (j)	Daily Staffing Hours (Input x_j)	Daily Revenue in \$100s (Output y_j)
Shop 1	10	20
Shop 2	15	45
Shop 3	12	24

Mathematical Model Formulation

Let $J = \{1,2,3\}$ represent the set of coffee shops. For each target shop $k \in J$ being evaluated, we solve a distinct optimization perspective within our system.

Parameters

- x_j : The input (Staffing Hours) for shop j
- y_j : The output (Revenue) for shop j
- ϵ : A tiny positive threshold (0.0001) to ensure weights do not drop to an unrealistic absolute zero.

Decision Variables

Because we want to solve for all shops simultaneously in a single execution run, we create an array of decision variables indexed by each evaluation perspective k :

- u_k : The output weight applied during the evaluation of shop k .
- v_k : The input weight applied during the evaluation of shop k .

Objective Function

We maximize the combined efficiency scores across all individual evaluations:

$$\max \quad Z = \sum_{k \in J} (u_k \cdot y_k)$$

CPLEX Code

```
performance analysis.mod ×
1 //
2
3
4 int numShops = 3;
5 range Shops = 1..numShops;
6 // Inputs: Daily Staffing Hours for Shop 1, 2, and 3
7 float inputs[Shops] = [10, 15, 12];
8 // Outputs: Daily Revenue (in $100s) for Shop 1, 2, and 3
9 float outputs[Shops] = [20, 45, 24];
10 // CHOOSE TARGET SHOP TO EVALUATE (Change this to 1, 2, or 3, then run)
11 int targetShop = 1;
12 // Tiny positive number to ensure weights stay above zero
13 float epsilon = 0.0001;
14 // MODEL DEFINITION
15 // Decision Variables (Weights)
16 dvar float+ u; // Output weight
17 dvar float+ v; // Input weight
18 // Objective: Maximize the efficiency score of our chosen target shop
19 maximize u * outputs[targetShop];
20 // Constraints
21 subject to {
22     // 1. Force the target shop's total weighted input to equal exactly 1
23     v * inputs[targetShop] == 1;
24     // 2. Ensure NO shop's efficiency can climb past 1.0 using these weights
25     forall(j in Shops) {
26         u * outputs[j] - v * inputs[j] <= 0;
27     }
28     // 3. Keep weights strictly positive
29     u >= epsilon;
30     v >= epsilon;
31 }
32 // OUTPUT LOG
33 execute {
34     writeln("");
35     writeln("--- Results for Shop ", targetShop, " ---");
36     writeln("Efficiency Score: ", cplex.getObjValue());
37     writeln("Output Weight (u): ", u);
38     writeln("Input Weight (v): ", v);
39     writeln("-----");
40     writeln("");
41 }
```

Result

```
Problems Scripting log Solutions × Conflicts Relaxations Engine log Statistics Profiler
// solution (optimal) with objective 0.666666666666667
// Quality There are no bound infeasibilities.
// There are no reduced-cost infeasibilities.
// Max. unscaled (scaled) Ax-b resid.          = 2.22045e-16 (6.93889e-18)
// Max. unscaled (scaled) c-B'pi resid.        = 1.77636e-15 (1.77636e-15)
// Max. unscaled (scaled) |x|                  = 0.1 (0.1)
// Max. unscaled (scaled) |slack|              = 0.4 (0.0999)
// Max. unscaled (scaled) |pi|                 = 0.666667 (14.2222)
// Max. unscaled (scaled) |red-cost|           = 0 (0)
// Condition number of scaled basis            = 4.7e+00
//
u = 0.0333333;
v = 0.1;
```

```
Problems Scripting log × Solutions Conflicts Relaxations Engine log
>> Scripting log (drop script code here to execute it)
// solution (optimal) with objective 0.666666666666667

--- Results for Shop 1 ---
Efficiency Score: 0.666666667
Output Weight (u): 0.033333333
Input Weight (v): 0.1
-----
```

Assignment

Your goal is to build a mathematical model in CPLEX OPL that evaluates all three shops simultaneously. The code must solve for the optimal weights of each shop and print a unified Performance Report Table displaying each shop's efficiency score and corresponding input/output weights.

Tips: // 1. Define sets and parameters // 2. Declare decision variables (arrays for u and v) // 3. Write the maximize objective function // 4. Construct subject to constraints (hint: use nested for all loops) // 5. Use an execute{} block to print the final data table

Experiment 7: Intermediate Optimization Modelling

Application (Inventory Control Problem)

Objectives

- To understand the dynamics of the multi-period Dynamic Lot-Sizing Problem (Wagner-Whitin model).
- To formulate a Mixed-Integer Linear Programming (MILP) model for balancing setup costs and inventory holding costs.
- To implement, execute, and verify an inventory optimization model using commercial optimization software.
- To interpret mathematical solution vectors (x, y, i) and translate them into actionable business logistics.

Software Used

- **IBM ILOG CPLEX Optimization Studio** (OPL Modelling Language and CPLEX Solver engine).

Introduction

In supply chain management, inventory control models determine *when* to order and *how much* to order. While elementary models like the Economic Order Quantity (EOQ) assume constant, continuous demand, real-world manufacturing environments deal with discrete timelines (weeks, months) where customer demand fluctuates heavily.

When demand varies by period, companies face a trade-off:

1. **Ordering frequently (Just-In-Time):** Minimizes holding costs because items spend less time in the warehouse, but dramatically increases Setup/Ordering Costs due to administrative and freight fees.
2. **Ordering in large bulk batches:** Minimizes setup costs by grouping multi-week demands into a single purchase order, but drastically increases Holding Costs because unsold inventory sits in storage accumulating fees.

To find the absolute lowest cost, we use Mixed-Integer Linear Programming (MILP). We introduce a binary "setup flag" variable ($y_t \in \{0,1\}$) that penalizes the system with a flat fee only when a purchase occurs, effectively linking discrete choices to continuous or integer product streams.

Problem Statement

A high-end electronics distribution center manages the logistics for a specific line of custom gaming laptops. The logistics manager needs to design a minimum-cost supply fulfillment plan over a 4-week planning horizon.

The operational parameters are defined as follows:

- Weekly Demand: Week 1 = 40 laptops, Week 2 = 60 laptops, Week 3 = 20 laptops, Week 4 = 80 laptops.
- Setup Cost (\$S\$): A flat \$1,000 freight handling and security fee is charged every time an order is placed, regardless of size.
- Unit Purchasing Cost (\$P\$): \$1,500 per laptop.
- Holding Cost (\$H\$): \$20 per laptop per week for insured warehouse storage.
- Initial Inventory: The warehouse begins Week 1 with 0 laptops.

Mathematical Model Formulation

Let t be the period index within the set $\text{Periods} = \{1, 2, 3, 4\}$.

$$\text{Minimize} \quad \sum_{t=1}^4 (\text{SetupCost} \cdot y_t + \text{UnitCost} \cdot x_t + \text{HoldingCost} \cdot i_t)$$

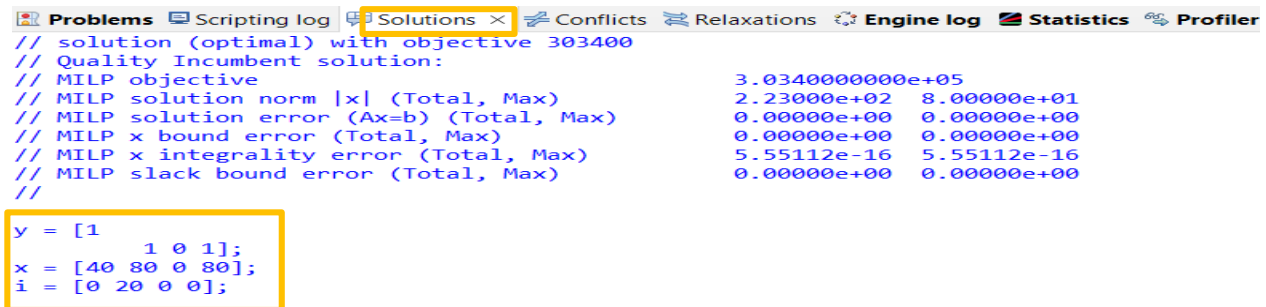
Subject to:

- *Inventory Balance (Period 1):* $x_1 - \text{Demand}_1 = i_1$
- *Inventory Balance (Periods 2 to 4):* $i_{t-1} + x_t - \text{Demand}_t = i_t \quad \forall t \in \{2, 3, 4\}$
- *Setup Linking Constraint:* $x_t \leq \text{BigM} \cdot y_t \quad \forall t \in \text{Periods}$
- *Variable Bounds:* $x_t, i_t \in \mathbb{Z}^+ \quad \text{and} \quad y_t \in \{0, 1\}$

CPLEX Code

```
*inv cont.mod ×
4 // 1. Parameters
5 int T = 4;
6 range Periods = 1..T;
7
8 int Demand[Periods] = [40, 60, 20, 80]; // Discrete integer demands
9 float SetupCost = 1000;
10 float UnitCost = 1500;
11 float HoldingCost = 20;
12
13 int BigM = sum(t in Periods) Demand[t];
14
15 // 2. Decision Variables (CHANGED TO int+ FOR CLEAN INTEGER RESULTS)
16 dvar int+ x[Periods]; // Discrete units ordered
17 dvar int+ i[Periods]; // Discrete units held in inventory
18 dvar boolean y[Periods]; // Binary setup flag
19
20 // 3. Objective Function
21 minimize
22     sum(t in Periods) (SetupCost * y[t] + UnitCost * x[t] + HoldingCost * i[t]);
23
24 // 4. Constraints
25 subject to {
26     x[1] - Demand[1] == i[1];
27
28     forall(t in 2..T) {
29         i[t-1] + x[t] - Demand[t] == i[t];
30     }
31     forall(t in Periods) {
32         x[t] <= BigM * y[t];
33     }
34 }
35 // 5. Post-Processing
36 execute {
37     writeln("--- OPTIMAL DISCRETE INVENTORY PLAN ---");
38     for(var t in Periods) {
39         writeln("Period ", t, ": Order Qty = ", x[t], ", Ending Inventory = ", i[t], ", Setup Flag = ", y[t]);
40     }
41 }
```

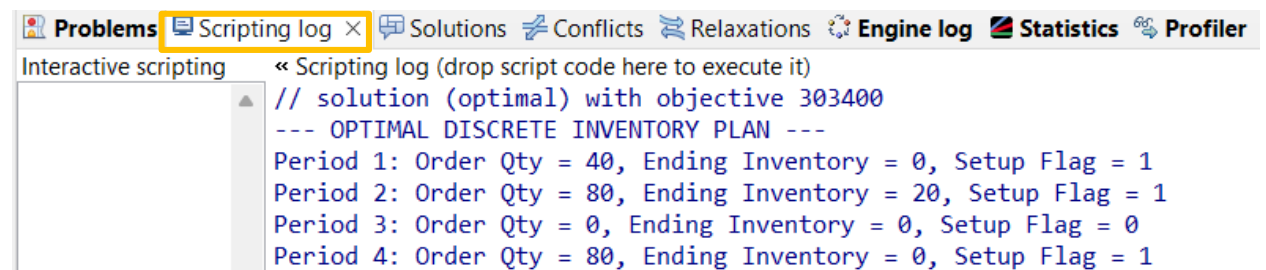
Result



The screenshot shows the 'Solutions' window of the CPLEX solver. It displays the optimal solution with an objective value of 303400. Below this, a table lists various metrics for the MILP solution, including the objective value, solution norm, solution error, bound error, integrality error, and slack bound error. A small code snippet is also visible in the bottom left corner of the window.

```
// solution (optimal) with objective 303400
// Quality Incumbent solution:
// MILP objective                                     3.0340000000e+05
// MILP solution norm |x| (Total, Max)               2.23000e+02  8.00000e+01
// MILP solution error (Ax=b) (Total, Max)           0.00000e+00  0.00000e+00
// MILP x bound error (Total, Max)                   0.00000e+00  0.00000e+00
// MILP x integrality error (Total, Max)              5.55112e-16  5.55112e-16
// MILP slack bound error (Total, Max)                0.00000e+00  0.00000e+00
//

y = [1
      1 0 1];
x = [40 80 0 80];
i = [0 20 0 0];
```



The screenshot shows the 'Scripting log' window of the CPLEX solver. It displays the optimal solution details, including the objective value and the optimal discrete inventory plan for four periods. The plan specifies the order quantity, ending inventory, and setup flag for each period.

```
« Scripting log (drop script code here to execute it)
// solution (optimal) with objective 303400
--- OPTIMAL DISCRETE INVENTORY PLAN ---
Period 1: Order Qty = 40, Ending Inventory = 0, Setup Flag = 1
Period 2: Order Qty = 80, Ending Inventory = 20, Setup Flag = 1
Period 3: Order Qty = 0, Ending Inventory = 0, Setup Flag = 0
Period 4: Order Qty = 80, Ending Inventory = 0, Setup Flag = 1
```

Assignment

A junior inventory controller suggests that placing only two orders across the whole month (ordering 120 units in Week 1 to cover weeks 1, 2, and 3, then ordering 80 units in Week 4) would be a superior strategy because it saves a \$1,000 setup fee by changing the setup array to

$$y = [1 \ 0 \ 0 \ 1].$$

Calculate the total cost of the junior controller's proposed schedule. Compare it against the CPLEX optimal objective value, and explain mathematically why human intuition failed to find the cheapest path.

Experiment 8: Advanced optimization modelling application (Facility Location Problem)

Objective

The objective of this experiment is to:

- Determine the lowest-cost combination of distribution centers by systematically evaluating all possible site combinations within the facility location problem.
- Implement a combinatorial search algorithm in MATLAB to compute and compare the total costs and coverage feasibility of all candidate facility subsets.
- Identify the global optimum by ranking all valid network configurations, thereby providing a verifiable proof that the selected solution achieves the absolute minimum expenditure.

Software

- Matlab

Introduction

The Facility Location Problem stands as a cornerstone of operations research and supply chain management, tasked with determining the most strategic spatial distribution of facilities to optimize service delivery. The primary objective is to minimize total capital expenditure and operational costs while simultaneously fulfilling a mandate of complete geographical coverage. In this context, decision-makers must evaluate a set of potential candidate sites against specific constraints, such as maximum allowable service distance, logistical capacity, or regional demand. Because the number of possible facility combinations grows exponentially as additional locations are added, the problem quickly evolves into a complex combinatorial search. When dealing with networks of manageable size, relying on intuition or manual trial-and-error becomes entirely insufficient, as the sheer volume of permutations obscures the most cost-effective solution. Consequently, an exhaustive search—a method where every feasible combination is systematically tested—becomes an ideal strategy. This approach is powerful because it does not merely estimate an answer; it evaluates the entire decision space, thereby providing objective, mathematical proof that the identified configuration is the absolute global minimum cost.

MATLAB serves as a premier computational platform for conducting this exhaustive analysis, offering sophisticated tools that bridge the gap between abstract mathematical formulation and actionable decision-making. By defining the decision space as a set of discrete variables, analysts can leverage MATLAB to programmatically iterate through every possible subset of facilities, checking each against the defined coverage constraints. This process effectively transforms opaque, multi-variable logistical problems into transparent, verifiable data sets. Beyond the raw calculation of costs, this programmatic approach allows users to perform sensitivity analyses—such as observing how the total cost changes if the service distance threshold is adjusted—which adds a layer of robustness to the decision-making process.

Problem

A company wants to serve 5 different regions (G, H, I, J, K) by establishing distribution centers. The goal is to cover all regions with the minimum possible total cost.

- **Coverage Distance (R):** 15 units. (A facility covers any region within a 15-unit distance).
- **Candidate Sites:** The regions themselves (G, H, I, J, K) are the potential facility locations.

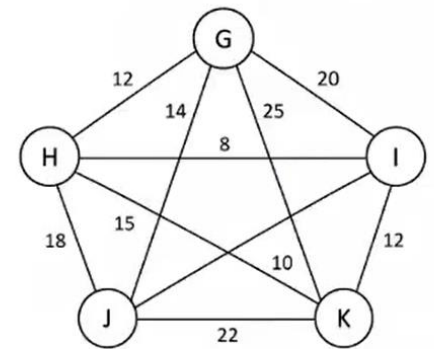
Inputs / Parameters

dij	G	H	I	J	K
G	0	12	20	14	25
H	12	0	8	18	15
I	20	8	0	10	12
J	14	18	10	0	22
K	25	15	12	22	0

aij	G	H	I	J	K
G	1	1	0	1	0
H	1	1	1	0	1
I	0	1	1	1	1
J	1	0	1	1	0
K	0	1	1	0	1

Fixed Costs for Locating a Facility (f_j):

- $f_G = 10 \mid f_H = 8 \mid f_I = 12 \mid f_J = 9 \mid f_K = 11$



Weighted graph based on distance matrix

Code

```
% --- Setup Data ---

costs = [10; 8; 12; 9; 11]; % G, H, I, J, K

A = [1 1 0 1 0; 1 1 1 0 1; 0 1 1 1 1; 1 0 1 1 0; 0 1 1 0 1];

siteNames = {'G', 'H', 'I', 'J', 'K'};


% --- Generate all possible combinations ---

allCombos = [];

for r = 1:5

    combos = nchoosek(1:5, r);

    % Pad with zeros to make them all 5 columns wide so they can be stacked

    padded = zeros(size(combos, 1), 5);

    padded(:, 1:r) = combos;

    allCombos = [allCombos; padded];

end


% --- Evaluate each combination ---

results = [];

for i = 1:size(allCombos, 1)

    % Get indices (ignore the padding zeros)

    combo = allCombos(i, allCombos(i,:) > 0);

    totalCost = sum(costs(combo));


    % Check if this combo covers all 5 regions

    isCovered = all(sum(A(:, combo), 2) > 0);

    if isCovered
```

```

        % Store: [Original Row Index, Total Cost]

        results = [results; i, totalCost];

    end

end

% --- Display Results Table ---

fprintf('%-20s | %-10s\n', 'Combination', 'Cost');

fprintf('-----\n');

[~, idx] = sort(results(:, 2));

sortedResults = results(idx, :);

for i = 1:min(10, size(sortedResults, 1))

    comboldx = sortedResults(i, 1);

    actualCombo = allCombos(comboldx, allCombos(comboldx,:) > 0);

    comboNames = strjoin(siteNames(actualCombo), ',');

    fprintf('%-20s | %-10d\n', comboNames, sortedResults(i, 2));

end

% --- Visualization ---

figure;

bar(sortedResults(1:min(10, end), 2));

ylabel('Total Cost'); title('Cheapest Valid Combinations');

grid on;

```

Assignment

Solve the same problem using the following modified values:

1. Coverage Distance (R): 12 units

2. Fixed Costs for Locating a Facility (f_j):

- $f_G = 12$
- $f_H = 7$
- $f_I = 10$
- $f_J = 11$
- $f_K = 9$

Experiment 9: Multi criteria and Goal programming techniques.

Objective

By the end of this experiment, students will be able to:

- Translate a set of conflicting or bounded requirements into a Goal Programming (GP) model using deviational variables.
- Distinguish between hard (system) constraints and soft (goal) constraints when building an optimization model.
- Implement and solve a GP model in Python using an open-source LP solver (PuLP / CBC).
- Critically evaluate whether Goal Programming is the appropriate modeling technique for a given problem.

Software Used:

- Python
- Gurobi/ PuLP

Background:

Goal Programming extends standard linear programming to situations with multiple, possibly conflicting, performance targets. Rather than optimizing a single objective function, each requirement is restated as a goal with a target value. Two non-negative deviational variables are attached to every goal:

- d^- — the amount by which the goal falls short of its target (underachievement)
- d^+ — the amount by which the goal exceeds its target (overachievement)

The objective function then minimizes only the deviations that are actually undesirable for each goal (e.g., underachievement for a minimum requirement, overachievement for a maximum requirement), rather than the raw performance measures themselves. Constraints that must hold exactly, with no acceptable deviation, remain ordinary hard constraints and receive no deviational variables.

Problem Statement

Circle K Farms must prepare exactly 3 tons (6,000 lb) of a special feed mix every day, blending three ingredients: limestone, corn, and soybean meal. The finished mix must meet the following nutritional requirements:

- Calcium content between 0.8% and 1.2% of total weight (a band, not a single target).
- Protein content of at least 22% of total weight.
- Fiber content of at most 5% of total weight.

The nutrient contribution of each ingredient (lb of nutrient per lb of ingredient) is given below:

Ingredient	Calcium	Protein	Fiber
Limestone	0.380	0.00	0.00
Corn	0.001	0.09	0.02
Soybean meal	0.002	0.50	0.08

Task: formulate this as a Goal Programming model, and form a reasoned opinion on whether GP is the right technique for this particular situation.

Mathematical Formulation

Decision Variables

- x_1 = pounds of limestone in the daily mix
- x_2 = pounds of corn in the daily mix
- x_3 = pounds of soybean meal in the daily mix

Hard Constraint

The total daily feed quantity is fixed — Circle K Farms always needs exactly 3 tons, so this requirement has no acceptable deviation and is left as an ordinary constraint, not a goal:

$$x_1 + x_2 + x_3 = 6000$$

Goal Constraints

Each percentage requirement is converted into a pound-target of the 6,000 lb mix (e.g., 0.8% of 6,000 lb = 48 lb). Note that the calcium band requires two separate goals — one for the floor, one for the ceiling — since it constrains both directions.

Goal	Requirement	Target (lb)	Goal Constraint
G1	Calcium $\geq 0.8\%$	48	$.380x_1 + .001x_2 + .002x_3 + d_1^- - d_1^+ = 48$
G2	Calcium $\leq 1.2\%$	72	$.380x_1 + .001x_2 + .002x_3 + d_2^- - d_2^+ = 72$
G3	Protein $\geq 22\%$	1320	$.09x_2 + .50x_3 + d_3^- - d_3^+ = 1320$
G4	Fiber $\leq 5\%$	300	$.02x_2 + .08x_3 + d_4^- - d_4^+ = 300$

Objective Function

Only the undesirable deviation direction is penalized for each goal — falling short of a minimum, or exceeding a maximum:

$$\min Z = d_1^- + d_2^+ + d_3^- + d_4^+$$

All decision and deviational variables are non-negative.

Procedure

1. Identify which requirements in the problem are true goals (can be traded off) versus hard constraints (cannot be violated). Justify your choice for the total-feed-weight requirement.
2. Derive the lb targets for each percentage requirement, using the 6,000 lb total.
3. Write the complete GP formulation (decision variables, hard constraint, goal constraints, objective function).
4. Implement the model in Python using PuLP (see reference implementation in Section 6) and solve it.
5. Record the optimal feed mix and the achieved value of each nutrient, and compare against the targets.

Reference Implementation (Python + PuLP)

PuLP is an open-source LP/MIP modeling library bundled with the CBC solver — no commercial license is required. Install with: `pip install pulp`

```
"""
Circle K Farms — Goal Programming Model (solved with PuLP / CBC, no Gurobi needed)
```

Decision variables (lb of ingredient in the daily 3-ton = 6000 lb mix):

x1 = limestone

x2 = corn

x3 = soybean meal

Hard constraint:

$x1 + x2 + x3 = 6000$ (total daily feed is fixed at 3 tons; this is not a goal)

Goals (nutrient requirements expressed as lb targets of the 6000 lb mix):

G1: Calcium $\geq 0.8\%$ $\rightarrow .380x1 + .001x2 + .002x3 \geq 48$ (goal, lower bound)

G2: Calcium $\leq 1.2\%$ $\rightarrow .380x1 + .001x2 + .002x3 \leq 72$ (goal, upper bound)

G3: Protein $\geq 22\%$ $\rightarrow .09x2 + .50x3 \geq 1320$ (goal, lower bound)

G4: Fiber $\leq 5\%$ $\rightarrow .02x2 + .08x3 \leq 300$ (goal, upper bound)

```
"""

from pulp import LpProblem, LpVariable, LpMinimize, lpSum, LpStatus, value
```

```
# ---- Model ----
```

```
model = LpProblem("CircleK_Farms_GP", LpMinimize)
```

```
# ---- Decision variables ----
```

```
x1 = LpVariable("Limestone_lb", lowBound=0)
```

```
x2 = LpVariable("Corn_lb", lowBound=0)
```

```
x3 = LpVariable("SoybeanMeal_lb", lowBound=0)
```

```

# ---- Deviation variables (one under- and one over-achievement var per goal) ----

d1_minus = LpVariable("d1_minus", lowBound=0) # calcium below 0.8% floor

d1_plus = LpVariable("d1_plus", lowBound=0)

d2_minus = LpVariable("d2_minus", lowBound=0)

d2_plus = LpVariable("d2_plus", lowBound=0) # calcium above 1.2% ceiling

d3_minus = LpVariable("d3_minus", lowBound=0) # protein below 22% floor

d3_plus = LpVariable("d3_plus", lowBound=0)

d4_minus = LpVariable("d4_minus", lowBound=0)

d4_plus = LpVariable("d4_plus", lowBound=0) # fiber above 5% ceiling


# ---- Hard constraint: total daily feed = 3 tons = 6000 lb ----

model += (x1 + x2 + x3 == 6000), "Total_Feed_Weight"


# ---- Goal constraints ----

calcium = 0.380 * x1 + 0.001 * x2 + 0.002 * x3

protein = 0.09 * x2 + 0.50 * x3

fiber = 0.02 * x2 + 0.08 * x3


model += (calcium + d1_minus - d1_plus == 48), "G1_Calcium_Min"

model += (calcium + d2_minus - d2_plus == 72), "G2_Calcium_Max"

model += (protein + d3_minus - d3_plus == 1320), "G3_Protein_Min"

model += (fiber + d4_minus - d4_plus == 300), "G4_Fiber_Max"


# ---- Objective: minimize only the undesirable deviations ----

# (falling short of minimums, exceeding maximums)

model += d1_minus + d2_plus + d3_minus + d4_plus


# ---- Solve ----

model.solve()


# ---- Report ----

print("Status:", LpStatus[model.status])

print("\n--- Optimal Feed Mix ---")

```

```

print(f"Limestone:   {x1.varValue:.2f} lb")

print(f"Corn:       {x2.varValue:.2f} lb")

print(f"Soybean meal: {x3.varValue:.2f} lb")

print(f"Total:      {x1.varValue + x2.varValue + x3.varValue:.2f} lb")


cal_val = value(calcium)

pro_val = value(protein)

fib_val = value(fiber)

print("\n--- Goal Achievement (as % of 6000 lb mix) ---")

print(f"Calcium: {cal_val:.2f} lb ({cal_val/6000*100:.3f}%) [target 0.8%-1.2%] "
      f"shortfall={d1_minus.varValue:.2f}, excess={d2_plus.varValue:.2f}")

print(f"Protein: {pro_val:.2f} lb ({pro_val/6000*100:.3f}%) [target >=22%] "
      f"shortfall={d3_minus.varValue:.2f}")

print(f"Fiber:   {fib_val:.2f} lb ({fib_val/6000*100:.3f}%) [target <=5%] "
      f"excess={d4_plus.varValue:.2f}")

print(f"\nTotal unwanted deviation (objective): {value(model.objective):.4f}")

```

Assignment:

- Mantel produces a toy carriage, whose final assembly must include four wheels and two seats. The factory producing the parts operates three shifts a day. The following table provides the amounts produced of each part in the three shifts:

Shift	Units produced per run	
	<i>Wheels</i>	<i>Seats</i>
1	500	300
2	600	280
3	640	360

Ideally, the number of wheels produced is exactly twice that of the number of seats.

However, because production rates vary from shift to shift, exact balance in production may not be possible. Mantel is interested in determining the number of production runs in each shift that minimizes the imbalance in the production of the parts. The capacity limitations

restrict the number of runs to between 4 and 5 for shift 1, 10 and 20 for shift 2, and 3 and 5 for shift 3. Formulate the problem as a GP model.

4. The Von Trapp family is in the process of moving to a new city where both parents have accepted new jobs. In trying to find an ideal location for their new home, the family list the following goals: (a) It should be as close as possible to Mrs. Von Trapp's place of work (within 1 4 mile). (b) It should be as far as possible from the noise of the airport (at least 10 miles). (c) It should be reasonably close to a shopping mall (within 1 mile). Mr. and Mrs. Von Trapp use a landmark in the city as a reference point and locate the (x, y)-coordinates of work, airport, and shopping mall at (1, 1), (20, 15), and (4, 7), respectively (all distances are in miles). Formulate the problem as a GP model.