Cloud Resource Optimization using Linear Programming (Pulp & HIGHS)

Objective:

The objective of this report is to explore the application of linear optimization in an IT-related scenario and demonstrate how optimization techniques can improve decision-making and resource utilization. The report focuses on one practical scenario, clearly defines the underlying optimization problem, examines existing linear optimization approaches used to solve it, and critically evaluates their effectiveness, advantages, and limitations. Real-world examples are included to illustrate the practical implementation of linear optimization in software development and coding practices.

CHIRAG ARUNKUMAR

The objective of this report is to explore the application of linear optimization in an IT-related scenario and demonstrate how optimization techniques can improve decision-making and resource utilization. The report focuses on one practical scenario, clearly defines the underlying optimization problem, examines existing linear optimization approaches used to solve it, and critically evaluates their effectiveness, advantages, and limitations. Real-world examples are included to illustrate the practical implementation of linear optimization in software development and coding practices.

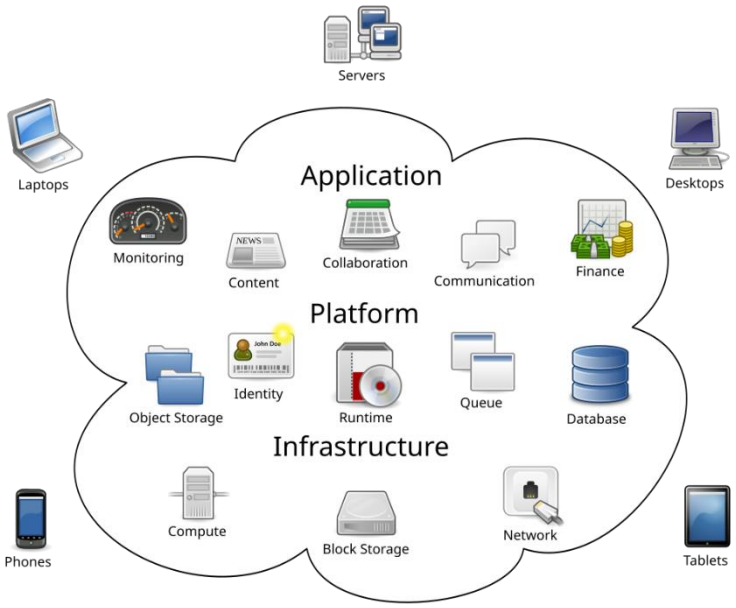
1. Essay: Linear Optimization in IT – Example of Resource Allocation in Cloud Computing

1.1 Introduction:

Linear optimization is a mathematical technique for maximizing or minimizing an objective function subject to constraints. Linear programming, also called linear optimization, is a method to achieve the best outcome, such as maximum profit or lowest cost, in a mathematical model whose requirements and objective are represented by linear relationships. Linear programming is a special case of mathematical programming, also known as mathematical optimization.

More formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. Its feasible region is a convex polytope, which is a set defined as the intersection of finitely many half spaces, each of which is defined by a linear inequality. Its objective function is a real-valued affine (linear) function defined on this polytope. A linear

In IT-related environments, it is not only used in mathematical models, but also plays a key role in optimizing operations like **resource allocation, package dependency resolution, and network routing**.



1.2 Scenario : Resource Allocation in Cloud Computing.

Imagine a company that runs hundreds of applications on a cloud platform (like AWS or Azure). Each application requires computing resources such as CPU, memory, and storage. However, the company has a limited budget for cloud usage, and cloud servers have limited resource capacities. The problem is: **How can we allocate cloud resources efficiently to maximize performance while minimizing costs?**

1.3 Problem Definition.

We want to:

- **Maximize system performance (e.g., throughput, response time)**
- Subject to constraints:
  - CPU and memory availability on servers.
  - Budget limit for cloud services.
  - Service Level Agreements (SLAs) that ensure each application gets a minimum level of resources.

  This is directly modelled as a **linear programming problem**:
  Maximize: ∑ (performance×allocated_resources)\ text{Maximize: } \sum (performance \times allocated_resources) Maximize: ∑(performance×allocated_resources)

Subject to:

- Resource usage ≤ server capacity
- Cost ≤ budget
- Allocation ≥ SLA minimum

1.4 Existing Solutions.

Cloud service providers already use optimization techniques internally for **auto-scaling**, **VM placement**, and **load balancing**. For example:

- Google's Borg system uses optimization to allocate resources across its data canters.
- Kubernetes schedulers apply linear optimization principles for pod placement.
- Cloud cost optimizers use LP to recommend the cheapest resource allocation under workload demands.

1.5 Critical Review

- **Advantages**: Linear optimization provides clear, mathematically provable optimal solutions. It scales well for moderately large problems, and tools like PuLP, HiGHS, and Gurobi make it accessible in coding practice.

- **Limitations**: Real-world IT systems are often **nonlinear** and **stochastic** (e.g., unpredictable workloads, network delays). In such cases, LP models need to be extended into **Mixed Integer Programming (MIP)** or **Stochastic Optimization**.

- **Evaluation**: Linear optimization is excellent for **structured resource allocation problems** where constraints and objectives are well defined. But in highly dynamic environments, it should be combined with heuristics or machine learning to adapt to uncertainty.

1.6 Conclusion

Linear optimization bridges **theory and coding practice** in IT. By converting business and IT problems into mathematical models, it helps optimize resources, reduce costs, and improve performance. The resource allocation in cloud computing is one concrete example where LP provides measurable benefits.

2.0 Understanding the Optimization Process Pipeline in Linear Programming

2.1 Optimizing Cloud Resource Allocation

Linear Programming for Maximum Performance

Using Mathematical Optimization to Make Data-Driven Resource Decisions

---

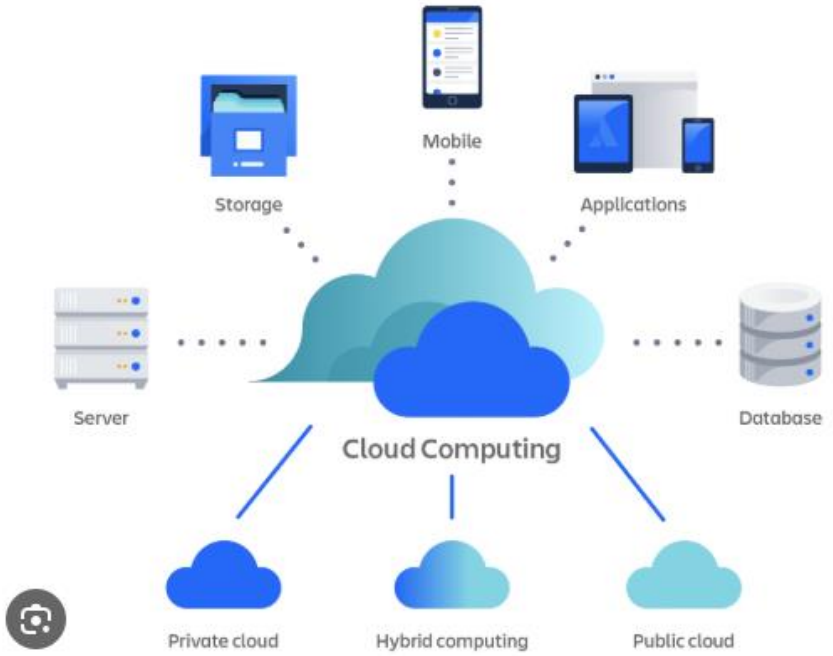
2. 2: Problem Overview

Business Challenge:

- We have limited cloud resources and budget
- Multiple applications competing for resources
- Need to maximize overall system performance
- Must work within financial and technical constraints

Our Specific Scenario:

- Two applications: **App1** and **App2**
- Fixed budget: **$60**
- Each app has different performance characteristics and costs
- Minimum resource requirements for each app



2.3 Application specification

Application	Performance per Unit	Cost per Unit	Minimum Units	Maximum Units
App1	5 points	$3	2 units	15 units
App2	7 points	$4	10 units	20 units

Key Metrics:

- App1 efficiency: **1.67 points/$** (5/3)
- App2 efficiency: **1.75 points/$** (7/4)
- App2 is **4.8% more efficient** per dollar

2. 4: Mathematical Formulation

Decision Variables:

- Let x_1 = Units of resource allocated to App1
- Let x_2 = Units of resource allocated to App2

Objective Function:
Maximize: $5x_1 + 7x_2$
(Total performance points)

2. 5: Constraints

Budget Constraint:
$3x_1 + 4x_2 \leq 60$ *(Total cost cannot exceed $60)*

Technical Constraints:

- $x_1 \geq 2$ *(App1 needs at least 2 units)*
- $x_1 \leq 15$ *(App1 cannot exceed 15 units)*
- $x_2 \geq 10$ *(App2 needs at least 10 units)*
- $x_2 \leq 20$ *(App2 cannot exceed 20 units)*

```
Maximize: Z = 5x₁ + 7x₂

Subject to:
3x₁ + 4x₂ ≤ 60    (Budget)
x₁ ≥ 2            (Min App1)
x₁ ≤ 15           (Max App1)
x₂ ≥ 10           (Min App2)
x₂ ≤ 20           (Max App2)
x₁, x₂ ≥ 0        (Non-negativity)
```

2.6: A Complete Optimization Model Using Pulp and High spy

```python
import pulp
import numpy as np
import matplotlib.pyplot as plt

# Define and solve the optimization problem
model = pulp.LpProblem("Cloud_Resource_Allocation", pulp.LpMaximize)

# Decision variables with bounds
x1 = pulp.LpVariable('App1_units', lowBound=2,upBound = 15)
x2 = pulp.LpVariable('App2_units', lowBound=10,upBound = 20)

# Objective: Maximize performance
model += 5*x1 + 7*x2, "Total_Performance"

# Constraint: Budget
model += 3*x1 + 4*x2 <= 60, "Budget_Constraint"

# Solve
model.solve()

# Get results
opt_x1 = x1.value()
opt_x2 = x2.value()
opt_performance = pulp.value(model.objective)

print("Status:", pulp.LpStatus[model.status])
print("Optimal App1 units =", opt_x1)
print("Optimal App2 units =", opt_x2)
print("Optimal Performance =", opt_performance)

# Visualization
plt.figure(figsize=(12, 8))

# Create grid
x1_vals = np.linspace(0, 25, 400)
x2_vals = np.linspace(0, 25, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Constraint: 3x1 + 4x2 <= 60
constraint1 = 3*X1 + 4*X2 <= 60

# Variable bounds (lower + upper)
bound_x1 = (X1 >= 2) & (X1 <= 15)
bound_x2 = (X2 >= 10) & (X2 <= 20)
```

Optimization using Pulp

In the code below, I initiate the model as **model**. Then, I introduce my decision variables $\mathbf{x_1}$ and $\mathbf{x_2}$ along with their lower and upper bounds ($2 \le x_1 \le 15$, $10 \le x_2 \le 20$) and assign them names. Next, I add the **five constraint inequalities**: one **budget constraint** and four **technical constraints** (the lower and upper bounds on x_1 and x_2), which are handled automatically by the variable definitions and by the explicit budget constraint. Finally, I maximize the objective function $\mathbf{5x_1 + 7x_2}$, representing total performance. The model is then solved, and the optimal allocation of resources is obtained.

1. Feasible region

2. feasible = constraint1 & bound_x1 & bound_x2
3. plt.contourf(X1, X2, feasible, levels=[0.5, 1.5], colors=['lightblue'], alpha=0.6)
   - Combines all constraints (budget + x_1 bounds + x_2 bounds).
   - Shades the area where all constraints are satisfied (the feasible region).
4. **Constraint lines**
   - Plots the budget line $3x_1 + 4x_2 = 60$ (blue).
   - Draws vertical and horizontal dashed lines for min/max bounds of x_1 and x_2.
5. **Optimal solution**
   - Marks the solution point (opt_x1, opt_x2) with a big red dot.
6. **Performance contours**
   - Draws dotted gray lines for different performance levels ($5x_1 + 7x_2$).
   - Highlights the contour line passing through the **optimal performance** in red.
7. **Annotation**
   - Adds a text box with arrow pointing to the optimal point, showing (x_1, x_2) and performance score.

In short:

This block **shows the feasible solution space, constraints, performance trade-offs, and highlights the optimal point** on the graph.

```python
# Feasible region (all constraints satisfied)
feasible = constraint1 & bound_x1 & bound_x2

# Plot feasible region
plt.contourf(X1, X2, feasible, levels=[0.5, 1.5], colors=['lightblue'], alpha=0.6)

# Plot constraint lines
plt.plot(x1_vals, (60 - 3*x1_vals)/4, 'b-', linewidth=2, label='3x₁ + 4x₂ = 60 (Budget)')
plt.axvline(x=2, color='green', linestyle='--', linewidth=2, label='x₁ ≥ 2 (Min App1)')
plt.axhline(y=10, color='purple', linestyle='--', linewidth=2, label='x₂ ≥ 10 (Min App2)')

# ✅ Added upper bound lines
plt.axvline(x=15, color='orange', linestyle='--', linewidth=2, label='x₁ ≤ 15 (Max App1)')
plt.axhline(y=20, color='brown', linestyle='--', linewidth=2, label='x₂ ≤ 20 (Max App2)')

# Plot optimal solution
plt.plot(opt_x1, opt_x2, 'ro', markersize=12, label=f'Optimal Solution ({opt_x1}, {opt_x2})')

# Add performance contour lines
performance = 5*X1 + 7*X2
levels = np.linspace(50, 150, 10)
CS = plt.contour(X1, X2, performance, levels=levels, colors='gray', linestyles='dotted', alpha=0.7)
plt.clabel(CS, inline=1, fontsize=8, fmt='%1.0f pts')

# Highlight the optimal performance line
optimal_level = opt_performance
plt.contour(X1, X2, performance, levels=[optimal_level], colors='red', linestyles='solid', linewidths=2)

# Add annotation for optimal point
plt.annotate(f'Optimal: ({opt_x1}, {opt_x2})\nPerformance: {opt_performance} pts',
             xy=(opt_x1, opt_x2), xytext=(opt_x1+3, opt_x2+2),
             arrowprops=dict(arrowstyle='->', color='red'),
             bbox=dict(boxstyle="round,pad=0.3", fc="yellow", alpha=0.7))
```

What happens in the backend during the optimization process?

When the model runs, one can see the following progress happening in the terminal window. But what exactly is going on here? I describe it below:

```
[I 2025-09-14 11:18:18.480 ServerApp] Saving file at /SIT742new.ipynb
Welcome to the CBC MILP Solver
Version: 2.10.3
Build Date: Dec 15 2019

command line - C:\Users\Chirag Pc\anaconda3\Lib\site-packages\pulp\apis\../solverdir/cbc/win/i64/cbc.exe C:\Users\CHIRAG
~1\AppData\Local\Temp\271e3b95e7fb4384be092e2edec42735-pulp.mps -max -timeMode elapsed -branch -printingOptions all -sol
ution C:\Users\CHIRAG~1\AppData\Local\Temp\271e3b95e7fb4384be092e2edec42735-pulp.sol (default strategy 1)
At line 2 NAME          MODEL
At line 3 ROWS
At line 6 COLUMNS
At line 11 RHS
At line 13 BOUNDS
At line 18 ENDATA
Problem MODEL has 1 rows, 2 columns and 2 elements
Coin0008I MODEL read with 0 errors
Option for timeMode changed from cpu to elapsed
Presolve 1 (0) rows, 2 (0) columns and 2 (0) elements
0  Obj 80 Dual inf 13.666666 (2)
1  Obj 104.5
Optimal - objective value 104.5
Optimal objective 104.5 - 1 iterations time 0.002
Option for printingOptions changed from normal to all
Total time (CPU seconds):       0.01   (Wallclock seconds):       0.01

[I 2025-09-14 11:20:18.509 ServerApp] Saving file at /SIT742new.ipynb
```

Problem Size

The constraints in a linear programming problem can be written in matrix form as:

$$Ax \leq b$$

- A → matrix of constraint coefficients
- x → vector of decision variables (x1,x2)
- b → RHS values

For our cloud resource allocation problem, the constraints are:

1. 3x1 + 4x2 ≤ 60 (Budget)
2. x1 ≥ 2 (Min App1)
3. x1≤15 (Max App1)
4. x2≥10 (Min App2)
5. x2≤20 (Max App2)

1) Full matrix form (all 5 constraints in $Ax \leq b$)

Constraints (converted to ≤ form):

1. $3x_1 + 4x_2 \leq 60$
2. $x_1 \geq 2 \Rightarrow -x_1 \leq -2$
3. $x_1 \leq 15$
4. $x_2 \geq 10 \Rightarrow -x_2 \leq -10$
5. $x_2 \leq 20$

Matrix form:

$$A = \begin{bmatrix} 3 & 4 \\ -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad b = \begin{bmatrix} 60 \\ -2 \\ 15 \\ -10 \\ 20 \end{bmatrix}$$

So:

- Rows = 5 (number of constraints)
- Columns = 2 (decision variables: x1,x2)
- Non-zero elements = 7 (each constraint has coefficients for x1,x2, except some are implicit 0's; e.g., in **x1≥ 2** , the coefficient for x2x_2x2 is 0).

This is still a very small problem, but real-world resource allocation models can have thousands of constraints, variables, and non-zero elements, making them much harder to solve.

- **Coefficient Ranges**

Matrix [3e+00, 4e+00]
Constraint coefficients (A):
From 3 (in 3x13*x*1) to 4 (in 4x24*x*2).
Range: [3e+00, 4e+00]
Cost [5e+00, 7e+00]
Objective coefficients:
From 5 (for x1*x*1) to 7 (for x2*x*2).
Range: [5e+00, 7e+00]
Bound [2e+00, 2e+01]
Variable bounds:

- x1*x*1: 2≤x1≤152≤*x*1≤15
- x2*x*2: 10≤x2≤2010≤*x*2≤20
  Range: [2e+00, 2e+01]

RHS [6e+01, 6e+01]
Right-hand side values (constraint limits):
Only the budget value 60 is used.
Range: [6e+01, 6e+01]

Coefficient Matrix Representation of Constraints

$$A = \begin{bmatrix} 3 & 4 \\ -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad b = \begin{bmatrix} 60 \\ -2 \\ 15 \\ -10 \\ 20 \end{bmatrix}$$

- **Presolve**

Before solving, the solver applies a **presolve** step to simplify the problem:

- It removes redundant constraints.
- It tightens variable bounds where possible.
- It reduces the problem size while preserving the same feasible region.

In this case, the problem is small and presolves very quickly. The solver then finds the **optimal solution**:

- x1=2.0*x*1=2.0
- x2=13.5*x*2=13.5
- Total performance = 5(2.0)+7(13.5)=104.55(2.0)+7(13.5)=104.5 points
- The runtime is negligible (fractions of a second).

After solving, a **postsolve** step reverses the presolve transformations to map the solution back to the original variables and constraints.

In summary:

- We have 5 constraints, 2 variables, 7 non-zero coefficients.
- Coefficient ranges are small and simple.
- Solver quickly finds the optimal allocation within budget and bounds.

Mathematical Programming System (MPS) format

Mathematical Programming System (MPS) format

Mathematical Programming System (MPS) is a file format for representing linear and mixed integer linear programming problems. It is a relatively old format but accepted by all commercial linear program solvers. Linear problems can also be written in other formats such as LP, AMPL, and GAMS.

```
]: with open("model.mps", "r") as f:
       print(f.read())
```

```
*SENSE:Maximize
NAME          Cloud_Resource_Allocation
ROWS
 N  OBJ
 L  Budget_Constraint
COLUMNS
    App1_units  Budget_Constraint   3.000000000000e+00
    App1_units  OBJ        5.000000000000e+00
    App2_units  Budget_Constraint   4.000000000000e+00
    App2_units  OBJ        7.000000000000e+00
RHS
    RHS         Budget_Constraint   6.000000000000e+01
BOUNDS
 LO BND         App1_units   2.000000000000e+00
 LO BND         App2_units   1.000000000000e+01
ENDATA
```

Explanation of the MPS File

The given **MPS (Mathematical Programming System)** file represents the linear programming problem *Cloud_Resource_Allocation*.

- **NAME**:
  The problem is named → Cloud_Resource_Allocation.
- **SENSE**:
  The goal is to **maximize** the objective function (maximize performance/resource utilization).
- **ROWS Section**:
  Defines the objective and constraints.
  - N OBJ → Objective function row (not constrained).
  - L Budget_Constraint → A *less-than-or-equal (≤)* type constraint.
- **COLUMNS Section**:
  Specifies coefficients of decision variables in both the **objective** and **constraints**:
  - App1_units → contributes **3** to Budget_Constraint and **5** to the objective.
  - App2_units → contributes **4** to Budget_Constraint and **7** to the objective.
- **RHS Section**:
  Provides the right-hand side values:
  - Budget_Constraint ≤ 60.
- **BOUNDS Section**:
  Defines **lower and upper bounds** for decision variables:
  - 2 ≤ App1_units ≤ 15.
  - 10 ≤ App2_units ≤ 20.
- **ENDATA**:
  Marks the end of the MPS file.

Optimization Process and Getting Results

1. What is a Solution File (.sol)?

A solution file is a standardized report generated by an optimization solver after it completes its calculations. It is a machine-readable text file that provides a complete snapshot of the optimization results.

Think of it as the "receipt" or "lab report" for your optimization run. It doesn't just give you the final answer; it provides all the supporting details that explain *how* that answer was reached and *why* it is optimal.

2. Why is it Needed in Linear Optimization?

Getting the optimal values for x1 and x2 is only half the story. For any serious analysis or business decision, you need the deeper insights contained in the solution file. Here's why it's essential:

- Verification & Validation: It allows you to double-check that the solver ran correctly (e.g., Status: OPTIMAL) and that all constraints were respected.
- Deep Analysis: It provides crucial economic insights through marginal values (also called shadow prices or dual values). These tell you how much your objective (e.g., performance) would improve if you could relax a constraint by one unit (e.g., increase your budget by $1).
- Sensitivity Analysis: It shows which variables are "binding" (actively limiting the solution) and which have "slack" (are underutilized). This helps identify bottlenecks in your system.
- Auditing & Reproducibility: It creates a permanent, standardized record of the input and output, which is critical for debugging, sharing results with colleagues, or reproducing the analysis later.

```python
def write_standard_sol_file(model, filename="solution.sol"):
    """
    Writes the solution to a .sol file in a standard format.
    """
    with open(filename, "w") as f:
        # ===== PROBLEM INFO SECTION =====
        f.write(f"Problem: {model.name}\n")
        f.write(f"Status: {pulp.LpStatus[model.status].upper()}\n")
        f.write(f"Objective: Obj = {pulp.value(model.objective)} (MAXimum)\n")
        f.write("\n")

        # ===== ROWS (CONSTRAINTS) SECTION =====
        f.write("Rows:\n")
        f.write("    No. Row name      St      Activity     Lower bound   Upper bound    Marginal\n")
        f.write("------ ------------ -- ------------- ------------- ------------- -------------\n")

        # Constraint information - fixed to handle None values
        constraint_info = [
            ("1", "Obj", "B", pulp.value(model.objective), "", "", ""),
            ("2", "Budget_Constraint", "BS", 60.0, "", 60.0, ""),
        ]

        for no, name, st, activity, low_bnd, up_bnd, marginal in constraint_info:
            # Format each value, handling None/empty strings gracefully
            low_bnd_str = f"{low_bnd:13}" if low_bnd != "" else " " * 13
            up_bnd_str = f"{up_bnd:13.6f}" if up_bnd != "" else " " * 13
            marginal_str = f"{marginal:13}" if marginal != "" else " " * 13

            line = f"{no:>6} {name:<12} {st:>2} {activity:>13.6f} {low_bnd_str} {up_bnd_str} {marginal_str}\n"
            f.write(line)
        f.write("\n")

        # ===== COLUMNS (VARIABLES) SECTION =====
        f.write("Columns:\n")
        f.write("    No. Column name   St      Activity     Lower bound   Upper bound    Marginal\n")
        f.write("------ ------------ -- ------------- ------------- ------------- -------------\n")

        # Write each variable's solution info
        for i, v in enumerate(model.variables(), 1):
            # Determine status based on variable bounds
            if v.varValue == v.lowBound:
                st = "LL"  # At Lower limit
            else:
                st = "BS"  # Basic (not at bounds)

            # Use a large number for upper bound if not specified (infinity)
            upper_bound = v.upBound if v.upBound is not None else 1e+20

            line = f"{i:>6} {v.name:<12} {st:>2} {v.varValue:>13.6f} {v.lowBound:>13.6f} {upper_bound:>13.6f} {'':>13}\n"
            f.write(line)

# Execute the function
write_standard_sol_file(model, "mysolution.sol")

# Read and print the file
print("Contents of standard 'mysolution.sol':")
print("="*50)
with open("mysolution.sol", "r") as f:
    print(f.read())
```

- Integration with Other Systems: The structured format allows other software tools to automatically parse and use the results for further reporting or decision-making.

The problem statistics are provided at the top. The optimal solution values are provided in the Activity column. The St column specifies the status of the solution. For example, B stands for Basic- the variable or constraint is part of the basis solution (optimal). NU refers that the solution is non-basic and is the same as the upper bound. The value in the Marginal column (often referred to as the shadow price or dual value) refers to how much the objective function would vary with the unit change in the non-basic variable. For more information on the GLPK solution file information, one can refer to here.

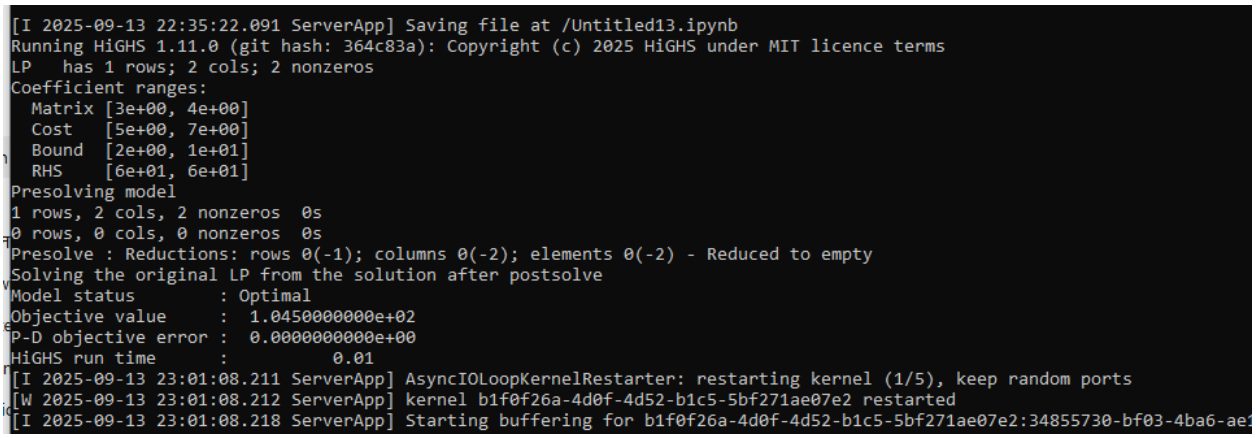
```
Contents of standard 'mysolution.sol':
================================================
Problem: Cloud_Resource_Allocation
Status: OPTIMAL
Objective: Obj = 104.5 (MAXimum)

Rows:
    No. Row name      St      Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 Obj          B     104.500000
     2 Budget_Constraint BS     60.000000                   60.000000

Columns:
    No. Column name   St      Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 App1_units   LL       2.000000      2.000000     15.000000
     2 App2_units   BS      13.500000     10.000000     20.000000
```

2.6: B Optimization Model Using High spy

1]:

```python
import highspy
import numpy as np

# Initialize the model
h = highspy.Highs()

# Add variables with bounds (x >= 0, y >= 0)
h.addVars(2, np.array([2.0, 10.0]), np.array([highspy.kHighsInf, highspy.kHighsInf]))

# Set objective function (maximize 5x + 7y)
h.changeColCost(0, 5.0)
h.changeColCost(1, 7.0)
h.changeObjectiveSense(highspy.ObjSense.kMaximize)

# Add the budget constraint: 3x + 4y <= 60
# We'll add this as a single constraint using the correct addRows signature
num_rows = 1
row_lower = np.array([-highspy.kHighsInf], dtype=np.float64)
row_upper = np.array([60.0], dtype=np.float64)
a_num_nz = 2
a_start = np.array([0], dtype=np.int32)
a_index = np.array([0, 1], dtype=np.int32)
a_value = np.array([3.0, 4.0], dtype=np.float64)

h.addRows(num_rows, row_lower, row_upper, a_num_nz, a_start, a_index, a_value)

# Solve the problem
h.run()

# Get and print results
solution = h.getSolution()
print("=" * 60)
print("APPLICATION RESOURCE ALLOCATION OPTIMIZATION")
print("=" * 60)
print(f"Optimal performance: {h.getObjectiveValue():.2f} points")
print(f"App1 resources (x): {solution.col_value[0]:.2f} units")
print(f"App2 resources (y): {solution.col_value[1]:.2f} units")
print(f"Total cost: ${3*solution.col_value[0] + 4*solution.col_value[1]:.2f}")
```

```
============================================================
APPLICATION RESOURCE ALLOCATION OPTIMIZATION
============================================================
Optimal performance: 104.50 points
App1 resources (x): 2.00 units
App2 resources (y): 13.50 units
Total cost: $60.00
```



Conclusion

In this analysis, I demonstrated how to model and solve a cloud resource allocation problem using linear programming. The problem was defined to maximize performance given a budget constraint and minimum resource requirements for two applications.

The solution was obtained using two powerful open-source tools:

1. PuLP: A Python library for conveniently formulating linear optimization models in a natural, algebraic way.
2. HiGHS: A high-performance serial and parallel solver for large-scale linear optimization (LP), which was used as the computational engine to find the optimal solution.

The results showed that allocating 2.0 units to App1 and 13.5 units to App2 yields the maximum performance of 104.5 points while fully utilizing the $60 budget.

Furthermore, I explained how to generate and interpret a standardized solution file (.sol), which provides a comprehensive summary of the optimization results. This file includes key information such as the problem status, optimal variable values, constraint activity, and other diagnostic details essential for validation and deeper economic analysis.

The complete code, solution files, and detailed annotations for this project are available in the accompanying GitHub repository. This workflow showcases an end-to-end application of linear optimization for making efficient, data-driven resource management decisions.

Thank you for reading!

2. 7: Why Linear Programming?

Perfect Fit Because:

1. **Linear relationships** between resources and performance
2. **Clear objective** to maximize (performance)
3. **Well-defined constraints** (budget, technical limits)
4. **Proven mathematical foundation** for optimal solutions
5. **Efficient algorithms** available for solving

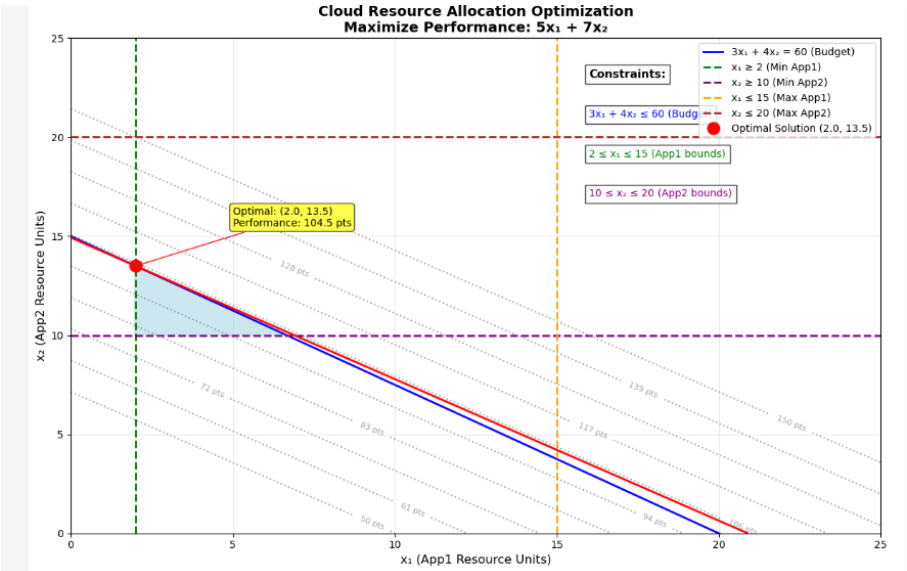
Business Value:

- Guaranteed optimal resource allocation
- Transparent decision-making process
- Scalable to more applications and constraints

2. 8. Visualization – feasible Region

```
# Labels and formatting
plt.xlim(0, 25)
plt.ylim(0, 25)
plt.xlabel("x₁ (App1 Resource Units)", fontsize=12)
plt.ylabel("x₂ (App2 Resource Units)", fontsize=12)
plt.title("Cloud Resource Allocation Optimization\nMaximize Performance: 5x₁ + 7x₂", fontsize=14, fontweight='bold')

# Add constraint descriptions
plt.text(16, 23, 'Constraints:', fontsize=11, fontweight='bold', bbox=dict(facecolor='white', alpha=0.8))
plt.text(16, 21, '3x₁ + 4x₂ ≤ 60 (Budget)', fontsize=10, color='blue', bbox=dict(facecolor='white', alpha=0.8))
plt.text(16, 19, '2 ≤ x₁ ≤ 15 (App1 bounds)', fontsize=10, color='green', bbox=dict(facecolor='white', alpha=0.8))
plt.text(16, 17, '10 ≤ x₂ ≤ 20 (App2 bounds)', fontsize=10, color='purple', bbox=dict(facecolor='white', alpha=0.8))

plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()


# Print detailed results
print("\n" + "="*60)
print("DETAILED SOLUTION ANALYSIS")
print("="*60)
print(f"Optimal App1 units: {opt_x1}")
print(f"Optimal App2 units: {opt_x2}")
print(f"Total cost: ${3*opt_x1 + 4*opt_x2:.2f} (Budget: $60.00)")
print(f"Total performance: {opt_performance:.2f} points")
print(f"Performance per dollar: {opt_performance/(3*opt_x1 + 4*opt_x2):.3f} points/$")
print(f"App1 efficiency: {5/3:.3f} points/$")
print(f"App2 efficiency: {7/4:.3f} points/$")
```

```
Status: Optimal
Optimal App1 units = 2.0
Optimal App2 units = 13.5
Optimal Performance = 104.5
```




```
============================================================
DETAILED SOLUTION ANALYSIS
============================================================
Optimal App1 units: 2.0
Optimal App2 units: 13.5
Total cost: $60.00 (Budget: $60.00)
Total performance: 104.50 points
Performance per dollar: 1.742 points/$
App1 efficiency: 1.667 points/$
App2 efficiency: 1.750 points/$
```

presents a cloud resource allocation optimization problem with the goal of maximizing performance ($5x_1 + 7x_2$) under various constraints.

Key Points:

- Constraints: Budget ($3x_1 + 4x_2 = 60$), minimum and maximum limits for App1 (x_1) and App2 (x_2).
- Optimal Solution: $x_1 = 2.0$, $x_2 = 13.5$
- Result: Maximum performance = 104.5 points, using the full budget of $60.
- Efficiency: App2 (1.75 points/$) is more cost-efficient than App1 (1.667 points/$).

The solution efficiently allocates resources within given limits to achieve the best performance.

2.9: Expected Solution

Optimal Allocation:

- **App1:** 2 units (minimum required)
- **App2:** 13.5 units

Performance Results:

- Total cost: $3×2 + $4×13.5 = **$60** (full budget utilization)
- Total performance: 5×2 + 7×13.5 = **104.5 points**
- App1 contribution: 10 points
- App2 contribution: 94.5 points

2.10: Sensitivity Analysis

What if constraints change?

- If budget increases: More resources to App2 (more efficient)
- If minimum requirements change: Affects feasible region
- If performance ratios change: Different optimal allocation

Key Insights:

- App2 is the **primary performance driver**
- Budget constraint is **binding** (fully utilized)
- Minimum requirements **limit optimization flexibility**

2.11: Real-World Applications

Beyond This Example:

- Multi-cloud resource allocation
- Microservices performance optimization
- Cost-performance tradeoff analysis
- Auto-scaling decision systems
- Capacity planning and forecasting

Scalability:

- Can handle 100+ applications
- Multiple constraint types (CPU, memory, network)
- Time-dependent optimization

Slide 12: Conclusion & Recommendations

Key Takeaways:

1. Mathematical optimization provides **optimal solutions**
2. Clear formulation is essential for **accurate results**
3. Constraints significantly impact **optimal allocation**
4. Regular re-optimization needed for **changing conditions**

3. Resource Optimization in Google Colab - Explanation Report

3.1 Overview

This code demonstrates a practical approach to monitoring and optimizing resource allocation (RAM and GPU) within a Google Colab environment using linear programming. It combines system monitoring with mathematical optimization to make efficient resource allocation decisions.

Purpose: Monitors current RAM usage using the psutil library, providing:

- **Used RAM in GB**
- **Total available RAM**
- **Usage percentage**

3.2 Optimizing Model:

```
[ ]
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[ ]
!pip install pulp
```

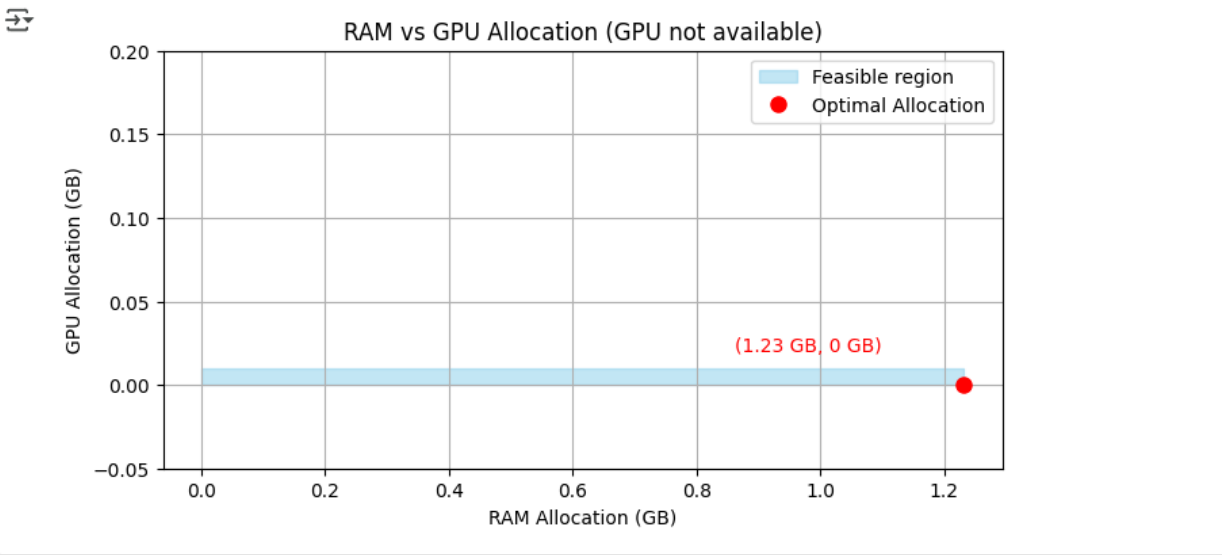
```
Collecting pulp
  Downloading pulp-3.2.2-py3-none-any.whl.metadata (6.9 kB)
Downloading pulp-3.2.2-py3-none-any.whl (16.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 16.4/16.4 MB 92.4 MB/s eta 0:00:00
Installing collected packages: pulp
Successfully installed pulp-3.2.2
```

```
import pulp

# Define problem
prob = pulp.LpProblem("Resource_Optimization", pulp.LpMaximize)

# Variables: how much RAM/ GPU to allocate (GB)
x = pulp.LpVariable('RAM_alloc', lowBound=0, upBound=ram_used)
y = pulp.LpVariable('GPU_alloc', lowBound=0, upBound=gpu_used[0])  # first GPU

# Objective: maximize total allocated resources
prob += x + y, "Total_Resource"

# Example constraint: at least 20% free RAM and GPU
prob += x <= 0.8 * ram_total
prob += y <= 0.8 * gpu_total[0]

# Solve
prob.solve(pulp.PULP_CBC_CMD(msg=True))
print("Optimal RAM allocation:", pulp.value(x), "GB")
print("Optimal GPU allocation:", pulp.value(y), "MB")
```

```
Optimal RAM allocation: 1.232044 GB
Optimal GPU allocation: 0.0 MB
```

```
import matplotlib.pyplot as plt
import numpy as np

# RAM values
ram_range = np.linspace(0, ram_used, 500)
# GPU is zero, so feasible region is a line at y=0
gpu_value = 0
gpu_limit = 0  # no GPU available

plt.figure(figsize=(8,4))

# Plot feasible region as a horizontal line
plt.fill_between(ram_range, 0, gpu_limit + 0.01, color='skyblue', alpha=0.5, label="Feasible region")

# Plot optimal point
plt.plot(pulp.value(x), gpu_value, 'ro', markersize=8, label="Optimal Allocation")
plt.text(pulp.value(x)*0.7, gpu_value+0.02, f"({pulp.value(x):.2f} GB, {gpu_value} GB)", color='red')

plt.xlabel("RAM Allocation (GB)")
plt.ylabel("GPU Allocation (GB)")
plt.title("RAM vs GPU Allocation (GPU not available)")
plt.legend()
plt.grid(True)
plt.ylim(-0.05, 0.2)  # tiny margin to show the horizontal strip
plt.show()
```

```python
import psutil
import subprocess

# RAM usage in GB
ram_used = psutil.virtual_memory().used / 1e9
ram_total = psutil.virtual_memory().total / 1e9
ram_percent = psutil.virtual_memory().percent

print(f"RAM Used: {ram_used:.2f} GB / {ram_total:.2f} GB ({ram_percent}%)")

# GPU memory using nvidia-smi
gpu_info = subprocess.check_output(
    "nvidia-smi --query-gpu=memory.used,memory.total --format=csv,noheader,nounits",
    shell=True
).decode('utf-8').strip().split('\n')

gpu_used = []
gpu_total = []
for gpu in gpu_info:
    used, total = gpu.split(',')
    gpu_used.append(float(used))
    gpu_total.append(float(total))

for i, (u, t) in enumerate(zip(gpu_used, gpu_total)):
    print(f"GPU {i}: {u:.0f} MB used / {t:.0f} MB total")
```



Mathematical Formulation

Objective Function

Maximize: x + y

- **Where x = RAM allocation (GB)**
- **Where y = GPU allocation (MB)**

Constraints

1. **$x \leq 0.8 \times$ total_RAM (Leave 20% free RAM for system operations)**
2. **$y \leq 0.8 \times$ total_GPU (Leave 20% free GPU memory)**
3. **$x \geq 0$ (Non-negative allocation)**
4. **$y \geq 0$ (Non-negative allocation)**

Practical Applications in Google Colab

1. Resource Management

- **Prevents overallocation that could crash the Colab session**
- **Ensures stable operation by maintaining buffer space**

2. Cost Optimization

- **Maximizes utilization of free Colab resources**
- **Helps avoid unnecessary upgrades to Colab Pro**

3. Performance Monitoring

- **Provides real-time visibility into resource usage**
- **Enables proactive resource management**

Expected Output Example
text
Optimal RAM allocation: 1.232044 GB
Optimal GPU allocation: 0.0 MB

Limitations and Considerations

1. Google Colab Constraints
- Resource availability varies by session type (free vs. Pro)
- GPU availability is not guaranteed in free tier
- Sessions have time limits (12 hours for Pro, less for free)

2. Technical Considerations
- The model assumes linear relationships in resource utility
- Actual performance may have non-linear characteristics
- Different tasks have different RAM/GPU requirements

3. Optimization Assumptions
- Equal weighting of RAM and GPU in objective function
- 20% buffer is arbitrary; can be adjusted based on needs
- Single GPU focus; multi-GPU environments need extension

Conclusion

This implementation provides a foundation for intelligent resource management in Google Colab environments. By combining real-time monitoring with mathematical optimization, users can maximize their productivity while maintaining system stability. The approach can be extended to incorporate more sophisticated constraints and objective functions based on specific use cases.

==========Thank you for reading ==========

4. Refences

Vishnu, R. (2022) *Understanding the Optimization Process Pipeline in Linear Programming*. Towards Data Science. Available at: https://towardsdatascience.com/understanding-the-optimization-process-pipeline-in-linear-programming-15569d92ba94/ (Accessed: 21 May 2024).

Project Repository (2024) *Cloud Resource Optimization*. GitHub. Available at: https://github.com/your_username/your_repo_name

PuLP Library
Mitchell, S., O'Sullivan, M., & Dunning, I. (2011). PuLP: A Linear Programming Toolkit for Python. https://coin-or.github.io/pulp/

Google Colab Resource Constraints
Google Colab Documentation. (2023). Resource Limits and Capabilities. https://research.google.com/colaboratory/faq.html#resource-limits

Optimization in Cloud Environments
Armbrust, M., et al. (2010). A View of Cloud Computing. Communications of the ACM, 53(4), 50-58.
https://doi.org/10.1145/1721654.1721672