Optimizing Packaging Material Usage for Multiple Items in a Single Box

1. Objectives and Introduction:

The goal and objective of this project is to find a way to minimize the material needed to create the surface area of a box that can fit multiple items of varying volumes without exceeding the volume of the box or the weight capacity of the box.

Lagrange multipliers were used to find the critical points of the box's width, length, and height which are represented as x, y, and z respectively. The goal was to find a constraint that allows the sum of the volumes of all items to be less than or equal to the volumes x*y*z of the box.

As this code is written in python, the user inputs dimensions of the items, their weights, and the weight capacity of the box. If the user enters weights greater than the weight capacity, the user is asked to re-input the weights.

2. Analytical Solution:

Assumptions:

It is assumed that the weights and volumes of 3 items are input by the user, each of which the volume is 10. Since the problem requires us to find the dimensions of a box that can perfectly fit items of this volume, it is assumed that the volume of the box will be greater than or equal to the combined volumes of the items. Hence the Lagrange constraint, x*y*z - (sum of volumes) = 0, meaning that there is a line of the form y = mx + b, passing through the origin under which the critical points of the box's dimensions shall be found.

Now that the constraint is defined, the function to find the critical points must also be defined. Luckily, as our requirement is to minimize the surface area of the box, the formula for the surface area of a cube/rectangle is used.

$f(x,y,z) = 2xy + 2yz + 2zx$

$g(x,y,z) = xyz - \text{sum of volumes} = 0$

Strategy:

Inputs:

- Box weight capacity.
- Number of Items.
- Volume of items (either same or varying - user decides).
- Weight of each item.

Input Validation:

- $xyz >= (\text{number of items} * (\text{volume of each item}))$
- weight of all items < Box weight capacity

Calculations:

$f(x, y, z) = 2xy + 2xz + 2yz$

$g(x, y, z) = xyz - (\text{number of items} * (\text{volume of each item})) >= 0$

$L(x, y, z) = f(x,y,z) - n * g(x,y,z)$

Find Partial Dervatives:

- $L_x = F_x - n(g_x)$
- $L_y = F_y - n(g_y)$
- $L_z = F_z - n(g_z)$

Find x, y and z in terms of m and n:

- $L_x = 0$
- $L_y = 0$
- $L_z = 0$

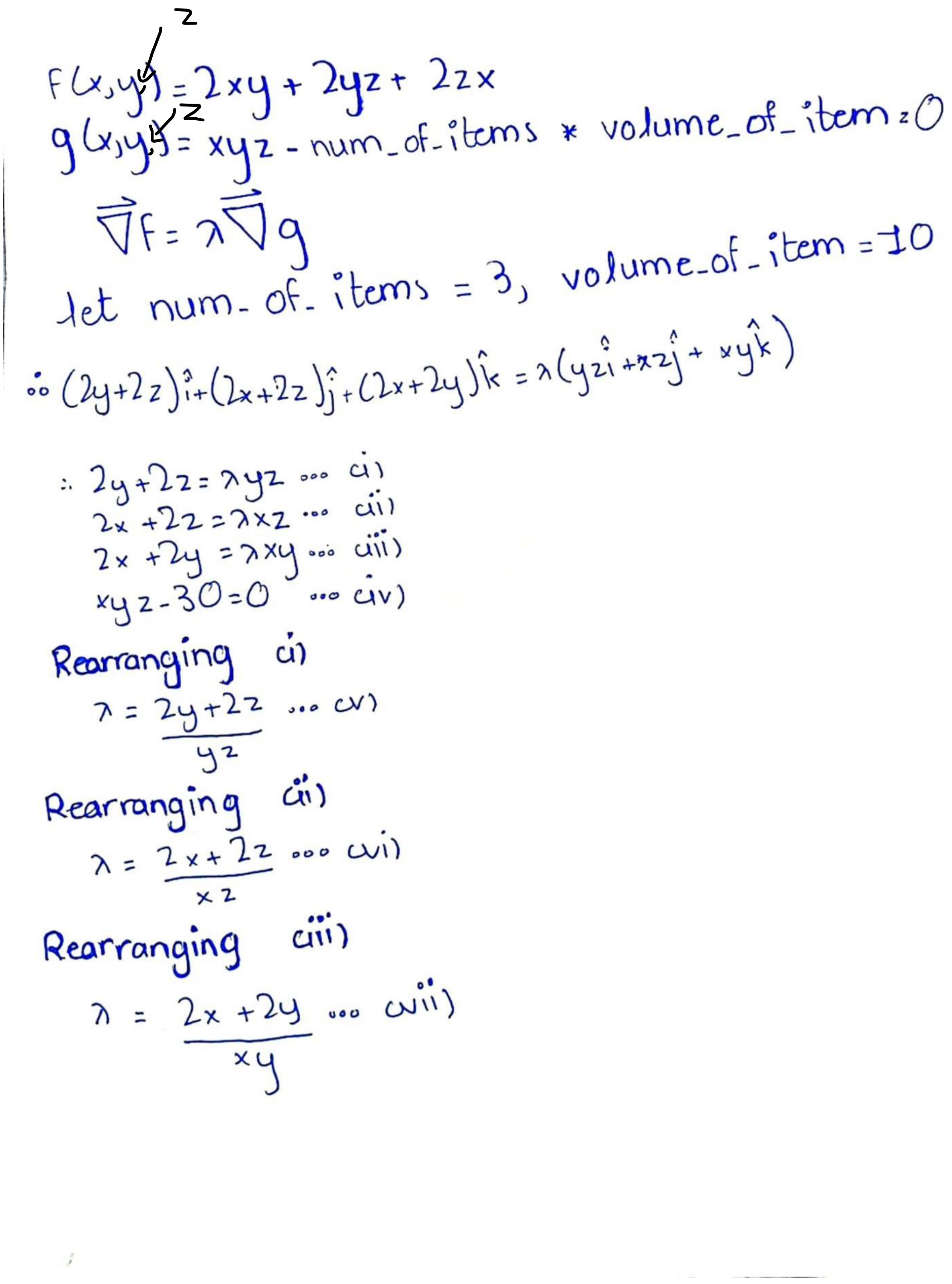
Find m and n:

- $g = 0$ ==> Put the values of x and y in this equation.

Find values of x, y and z by replacing m and n:

- $L_x = 0$
- $L_y = 0$
- $L_z = 0$

Analytical Solution Steps:



$F(x,y\overset{z}{\swarrow}) = 2xy + 2yz + 2zx$

$g(x,y\overset{z}{\swarrow}) = xyz$ - num_of_items * volume_of_item $= 0$

$\vec{\nabla} f = \lambda \vec{\nabla} g$

let num_of_items $= 3$, volume_of_item $= 10$

$\therefore (2y+2z)\hat{i} + (2x+2z)\hat{j} + (2x+2y)\hat{k} = \lambda(yz\hat{i} + xz\hat{j} + xy\hat{k})$

$\therefore 2y+2z = \lambda yz \quad \ldots \text{(i)}$

$2x+2z = \lambda xz \quad \ldots \text{(ii)}$

$2x+2y = \lambda xy \quad \ldots \text{(iii)}$

$xyz-30=0 \quad \ldots \text{(iv)}$

Rearranging (i)

$\lambda = \frac{2y+2z}{yz} \quad \ldots \text{(v)}$

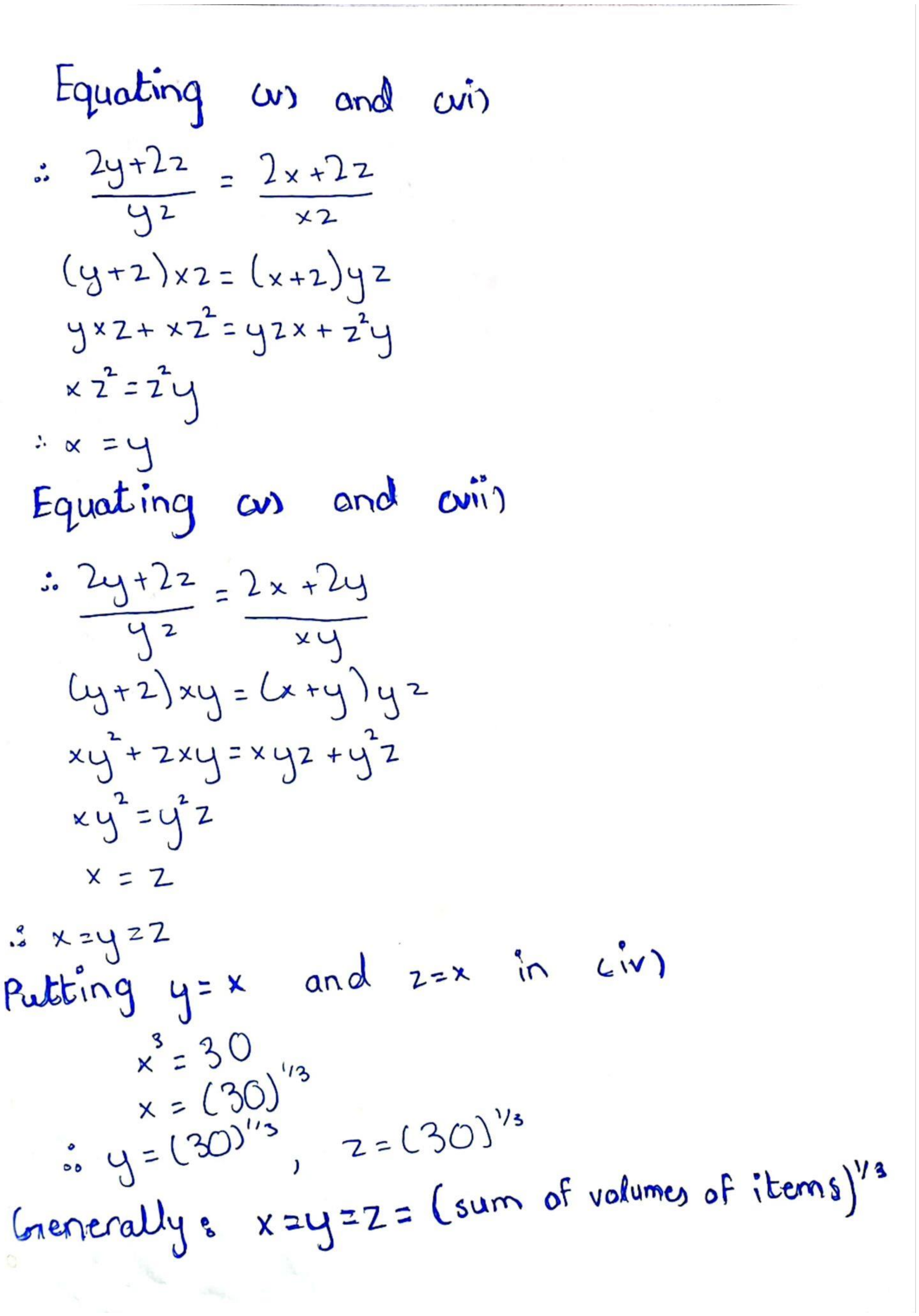
Rearranging (ii)

$\lambda = \frac{2x+2z}{xz} \quad \ldots \text{(vi)}$

Rearranging (iii)

$\lambda = \frac{2x+2y}{xy} \quad \ldots \text{(vii)}$

Hand-written solution



Equating (v) and (vi)

$$\therefore \frac{2y+2z}{yz} = \frac{2x+2z}{xz}$$

$$(y+z)xz = (x+z)yz$$

$$yxz + xz^2 = yzx + z^2y$$

$$xz^2 = z^2y$$

$$\therefore x = y$$

Equating (v) and (vii)

$$\therefore \frac{2y+2z}{yz} = \frac{2x+2y}{xy}$$

$$(y+z)xy = (x+y)yz$$

$$xy^2 + zxy = xyz + y^2z$$

$$xy^2 = y^2z$$

$$x = z$$

$$\therefore x = y = z$$

Putting $y = x$ and $z = x$ in (iv)

$$x^3 = 30$$

$$x = (30)^{1/3}$$

$$\therefore y = (30)^{1/3}, \quad z = (30)^{1/3}$$

Generally: $x = y = z = (\text{sum of volumes of items})^{1/3}$

Hand-Written solution

3. Python Code:

```
import sympy as sp  # Importing SymPy library for symbolic mathematics
import numpy as np  # Importing NumPy library for numerical computations
import matplotlib.pyplot as plt  # Importing matplotlib library for plotting
from mpl_toolkits.mplot3d import Axes3D  # Importing Axes3D module for 3D plotting

# Initialize pretty printing for SymPy
sp.init_printing()

# Define the variables for the box dimensions
x, y, z = sp.symbols('x y z', real=True, positive=True)

# Taking inputs from the user
box_weight_capacity = int(input("Enter the Box weight capacity: "))  # Requesting box
weight capacity from the user
num_items = int(input("Enter the Number of Items: "))  # Requesting the number of items
from the user

items = []  # Initializing an empty list to store item details
total_weight = 0  # Initializing total weight of items to 0
total_volume = 0  # Initializing total volume of items to 0

# Flag to control the input loop
isFalse = False
while not isFalse:
    # Loop to iterate over each item and get its weight and volume
    for i in range(num_items):
        weight = int(input("Enter the weight of item " + str(i+1) + ": "))  # Requesting
weight of each item
        volume = int(input("Enter the volume of item " + str(i+1) + ": "))  # Requesting
volume of each item
        total_weight += weight  # Adding weight of current item to total weight
        total_volume += volume  # Adding volume of current item to total volume
        items.append((weight, volume))  # Storing weight and volume of item as a tuple in
the list

    # Checking if the total weight of items is within box weight capacity
    if box_weight_capacity - total_weight >= 0:
        isFalse = True  # If yes, exit the loop
    else:
        print("Weight capacity exceeded, please provide correct inputs!!!")  # If not,
prompt the user to provide correct inputs
        total_weight = 0  # Reset total weight to 0
        total_volume = 0  # Reset total volume to 0
        items.clear()  # Clear the list of items

# Define the function to optimize and the constraints
f = 2*x*y + 2*x*z + 2*y*z  # Function representing the surface area of the box
g1 = x*y*z - total_volume  # Constraint representing the total volume of items
```

```
# Define the Lagrange multiplier
n = sp.symbols('n', real=True)

# Define the Lagrange function
L = f - n * g1

# Calculate the partial derivatives
eq1 = sp.diff(L, x)  # Partial derivative with respect to x
eq2 = sp.diff(L, y)  # Partial derivative with respect to y
eq3 = sp.diff(L, z)  # Partial derivative with respect to z
eq4 = g1  # The constraint equation

# Solve the system of equations
solution = sp.solve([eq1, eq2, eq3, eq4], [x, y, z, n], dict=True)

# Display the real solutions
for sol in solution:
    if all(value.is_real for value in sol.values()):
        x_val = sol[x].evalf()  # Evaluate x
        y_val = sol[y].evalf()  # Evaluate y
        z_val = sol[z].evalf()  # Evaluate z
        lambda_val = sol[n].evalf()  # Evaluate lambda
        print(f"Solution:\nx = {x_val}\ny = {y_val}\nz = {z_val}\nn = {lambda_val}")

        # Plot the main storage box (edges only)
        fig_main = plt.figure()
        ax_main = fig_main.add_subplot(111, projection='3d')

        # Define the vertices of the main box
        main_box_vertices = np.array([[0, 0, 0], [x_val, 0, 0], [x_val, y_val, 0], [0,
y_val, 0],
                                      [0, 0, z_val], [x_val, 0, z_val], [x_val, y_val,
z_val], [0, y_val, z_val]])

        # Define the edges of the main box
        main_box_edges = [
            [main_box_vertices[0], main_box_vertices[1], main_box_vertices[2],
main_box_vertices[3], main_box_vertices[0]],
            [main_box_vertices[4], main_box_vertices[5], main_box_vertices[6],
main_box_vertices[7], main_box_vertices[4]],
            [main_box_vertices[0], main_box_vertices[4]], [main_box_vertices[1],
main_box_vertices[5]],
            [main_box_vertices[2], main_box_vertices[6]], [main_box_vertices[3],
main_box_vertices[7]]
        ]

        # Plot the edges of the main box
        for edge in main_box_edges:
            x_edge, y_edge, z_edge = zip(*edge)
            ax_main.plot(x_edge, y_edge, z_edge, color='b')
        # Set axis labels and title for the main box plot
        ax_main.set_xlabel('X')
        ax_main.set_ylabel('Y')
        ax_main.set_zlabel('Z')
        ax_main.set_title('Main Storage Box (Edges Only)')

        # Calculate the maximum side length of each item box
        max_item_side_length = sp.cbrt(max(volume for _, volume in items))

        # Calculate the number of item boxes in each dimension
        num_boxes_per_side = sp.ceiling(sp.cbrt(num_items))
        num_boxes = num_boxes_per_side ** 3

        # Plot the smaller item boxes (edges only)
        fig_items = plt.figure()
        ax_items = fig_items.add_subplot(111, projection='3d')

        # Plot the edges of each item box
        colors = plt.cm.rainbow(np.linspace(0, 1, num_items))
        idx = 0
        for i in range(num_boxes_per_side):
            for j in range(num_boxes_per_side):
                for k in range(num_boxes_per_side):
                    if idx < num_items:
                        x_item = i * max_item_side_length
                        y_item = j * max_item_side_length
                        z_item = k * max_item_side_length
                        weight, volume = items[idx]
                        item_side_length = sp.cbrt(volume)
                        # Define the vertices of the item box
                        item_box_vertices = np.array([[x_item, y_item, z_item],
                                                      [x_item + item_side_length, y_item,
z_item],
                                                      [x_item + item_side_length, y_item +
item_side_length, z_item],
                                                      [x_item, y_item + item_side_length,
z_item],
                                                      [x_item, y_item, z_item +
item_side_length],
                                                      [x_item + item_side_length, y_item,
z_item + item_side_length],
                                                      [x_item + item_side_length, y_item +
item_side_length, z_item + item_side_length],
                                                      [x_item, y_item + item_side_length,
z_item + item_side_length]])

                        # Define the edges of the item box
                        item_box_edges = [
                            [item_box_vertices[0], item_box_vertices[1],
item_box_vertices[2], item_box_vertices[3], item_box_vertices[0]],
                            [item_box_vertices[4], item_box_vertices[5],
item_box_vertices[6], item_box_vertices[7], item_box_vertices[4]],
                            [item_box_vertices[0], item_box_vertices[4]],
[item_box_vertices[1], item_box_vertices[5]],
                            [item_box_vertices[2], item_box_vertices[6]],
[item_box_vertices[3], item_box_vertices[7]]
                        ]

                        # Plot the edges of the item box with transparency
                        for edge in item_box_edges:
                            x_edge, y_edge, z_edge = zip(*edge)
                            ax_items.plot(x_edge, y_edge, z_edge, color=colors[idx],
alpha=0.5)

                        idx += 1

        # Set axis labels and title for the item boxes plot
        ax_items.set_xlabel('X')
        ax_items.set_ylabel('Y')
        ax_items.set_zlabel('Z')
        ax_items.set_title('Item Boxes (Edges Only)')

        # Calculate the maximum dimension among all axes
        max_dim = max(x_val, y_val, z_val, max_item_side_length * num_boxes_per_side)

        # Set limits for each axis to the maximum value among all axes
        ax_main.set_xlim([0, float(max_dim)])
        ax_main.set_ylim([0, float(max_dim)])
        ax_main.set_zlim([0, float(max_dim)])

        ax_items.set_xlim([0, float(max_dim)])
        ax_items.set_ylim([0, float(max_dim)])
        ax_items.set_zlim([0, float(max_dim)])

plt.show()  # Display the plots
```

Explanation:

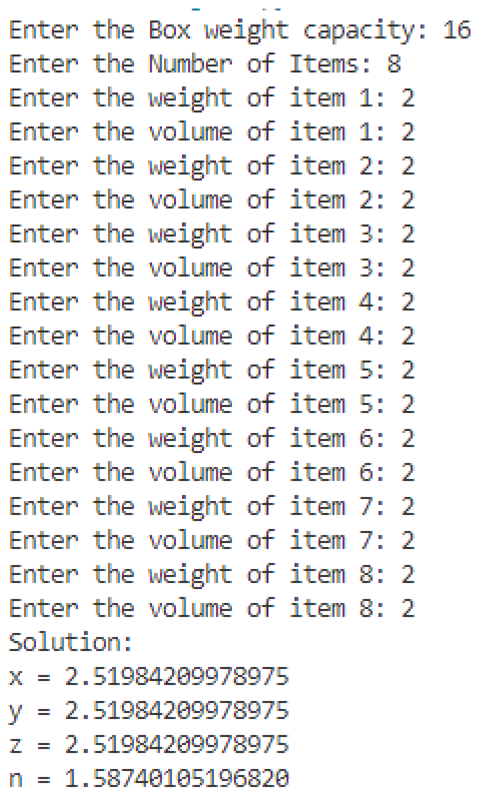
This Python code prompts the user to input the weight capacity of a storage box and the number of items to be stored inside it. It then prompts for the weight and volume of each item. The program optimizes the arrangement of the items inside the box to maximize the total surface area of the box while satisfying the volume constraint using the method of Lagrange multipliers.

The main steps of the code include:

- Importing necessary libraries: seemly, numpy, and matplotlib.
- Getting user input for box weight capacity and the number of items.

- Iterating through each item to get its weight and volume, checking if the total weight does not exceed the box weight capacity.
- Defining the function to optimize (surface area of the box) and the constraint (total volume of items).
- Solving the optimization problem using the method of Lagrange multipliers to find the dimensions of the box.
- Plotting the main storage box and the smaller item boxes using matplotlib's 3D plotting capabilities.
- Calculating the maximum side length of each item box and the number of item boxes per side.
- Plotting the smaller item boxes inside the main box, ensuring no overlap, and maintaining proper positioning.
- Setting appropriate axis labels, titles, and limits to ensure proper visualization.
- Displaying the plots using plt.show().

4. Python Solution and Results:



```
Enter the Box weight capacity: 16
Enter the Number of Items: 8
Enter the weight of item 1: 2
Enter the volume of item 1: 2
Enter the weight of item 2: 2
Enter the volume of item 2: 2
Enter the weight of item 3: 2
Enter the volume of item 3: 2
Enter the weight of item 4: 2
Enter the volume of item 4: 2
Enter the weight of item 5: 2
Enter the volume of item 5: 2
Enter the weight of item 6: 2
Enter the volume of item 6: 2
Enter the weight of item 7: 2
Enter the volume of item 7: 2
Enter the weight of item 8: 2
Enter the volume of item 8: 2
Solution:
x = 2.51984209978975
y = 2.51984209978975
z = 2.51984209978975
n = 1.58740105196820
```

Output to the console of the program with certain user inputs

The program allows the user to enter the box weight capacity, number of items and then the weight and volume of each item.

During the input process, the program validates the inputs making sure that the total item weight does not exceed the weight limit of the main storage box.

Then the objective function represting the surface area of the box and the constraint function represting the total volume of items are defined. Called f, and g1 respectively.

f = 2xy + 2xz + 2yz
g1 = xyz - total_volume
where total volume is the sum of the volumes of the items.

The lagrange multipler is defined as ‘n’ (Known in the analytical solution as delta Δ). Then the Lagrange function is defined as L = f – n * g1 (Equal to Δf = Δg in the analytical solution)

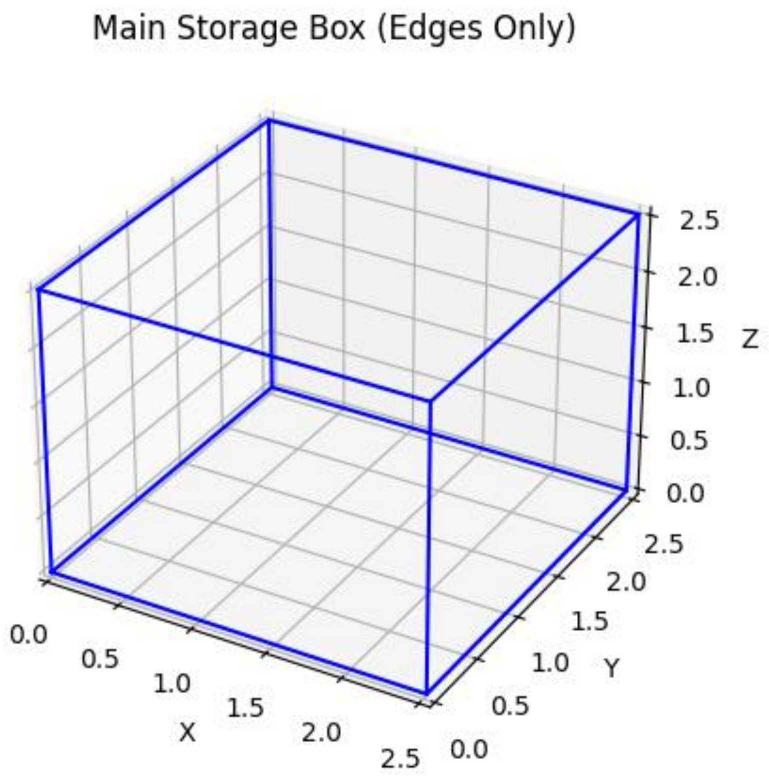
The system of equations then required to solve the problem is built by defining eq1, eq2, eq3 which are partial derivatives with respect to x, y, and z, respectively. The fourth and final equation eq4 is simply the constraint equation g itself. Like the analytical solution in which the system of equations is solved manually.

After solving the system of equations, we filter out the real solutions and evaluate the symbolic representation of x, y and z and store them in x_val, y_val, and z_val, and print them to the console. The same is done for n.

So far, the analytical solution has kept up with the python solution, however; from this point onward, as we progress to plotting the solutions the analytical solution stops here as it does not discuss the graphs.

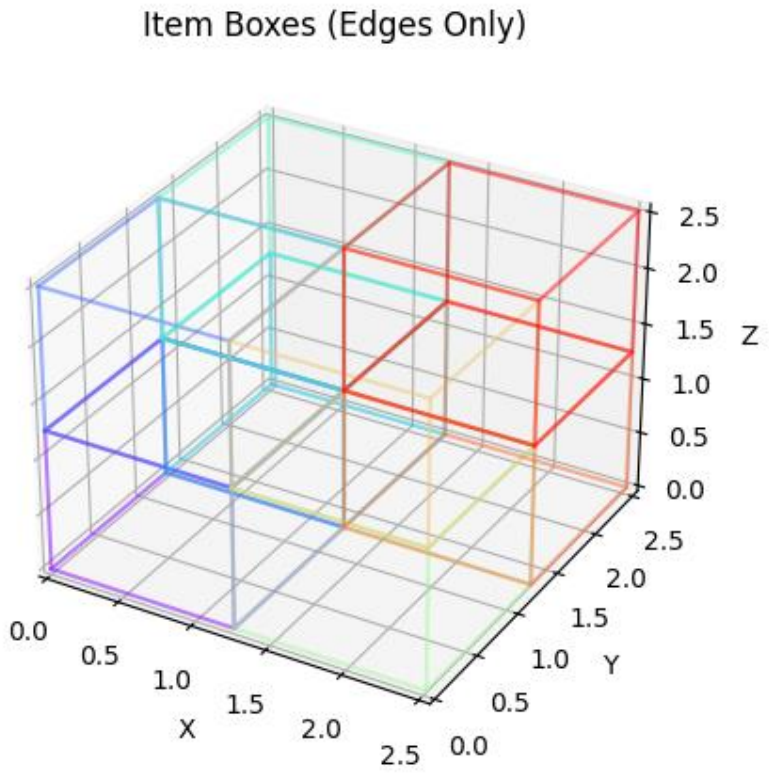
The process for plotting the graphs of the results is as follows, x_val, y_val, z_val are all interpreted as float values, then these are stored as vertices in a np.array and from that, they are packed into edges represented by two vertices each.

For each edge, the information about each point of the edge is unpacked and stored into x_edge, y_edge, and z_edge, then this is plotted.



A plot of the main box

The same process is repeated for the item boxes, but in their case, the number of boxes, the length of a side of a box, and the position that they should be plotted in is carefully noted in the code. The vertices of each box is defined with the help of the loops and variables x_item, y_item, z_item, which store the position at which the item boxes should be drawn from, and item_side_length, which stores the length of a side of the item, this was calculated by taking the cube root of the volume of that item.

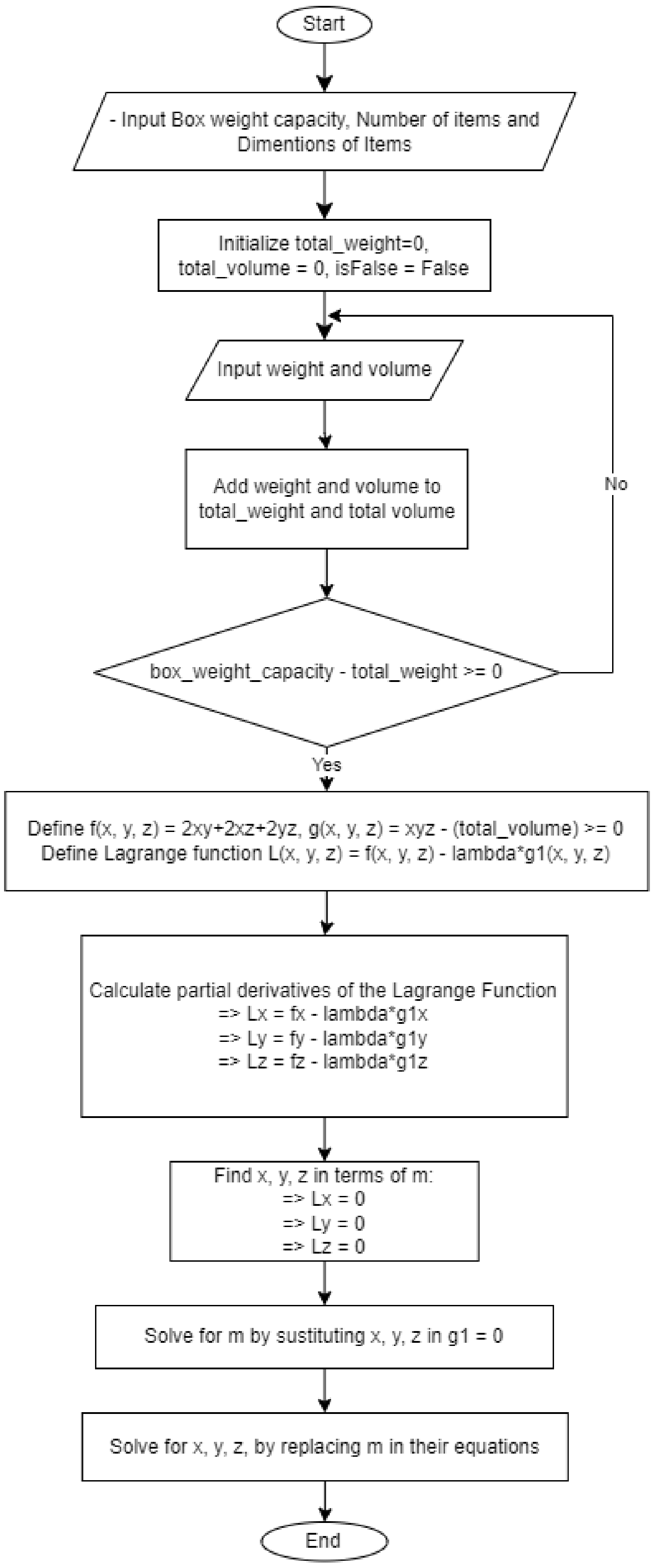


A plot of the items

For both plots, the maximum value on any axis is found and then the limits of the graphs are determined so that the final graphs are of proper scale.

The physical interpretation of the results is straightforward, the final value of x, y, and z, represent the length, width, and height of the packacging box respectively. The secondary plot of item boxes represents the possible arrangements of the items within the main packaging box.

5. Flow Chart:


Start
- Input Box weight capacity, Number of items and Dimentions of Items
Initialize total_weight=0, total_volume = 0, isFalse = False
Input weight and volume
Add weight and volume to total_weight and total volume
box_weight_capacity - total_weight >= 0
No
Yes
Define f(x, y, z) = 2xy+2xz+2yz, g(x, y, z) = xyz - (total_volume) >= 0
Define Lagrange function L(x, y, z) = f(x, y, z) - lambda*g1(x, y, z)
Calculate partial derivatives of the Lagrange Function
=> Lx = fx - lambda*g1x
=> Ly = fy - lambda*g1y
=> Lz = fz - lambda*g1z
Find x, y, z in terms of m:
=> Lx = 0
=> Ly = 0
=> Lz = 0
Solve for m by sustituting x, y, z in g1 = 0
Solve for x, y, z, by replacing m in their equations
End

Flowchart of the methodology used in the model

6. Conclusions:

This project aimed to optimize packaging material usage for multiple items within a single box by leveraging Lagrange multipliers and Python programming. The problem was well-defined, emphasizing the need to minimize material usage while adhering to weight capacity constraints. Through systematic implementation of the solution methodology, including user input validation, function definition, and symbolic equation solving, the Python program successfully generated optimal dimensions for the storage box and visualized the arrangement of items within it.

Key insights from the project include:

The use of Lagrange multipliers proved effective in finding optimal solutions, demonstrating the applicability of advanced mathematical techniques in logistics optimization.

Python programming facilitated the implementation of the solution, enabling efficient computation and visualization of results.

The results obtained showcased the potential for significant material savings and highlighted the importance of considering weight capacity constraints in packaging design.

Overall, this project underscores the value of mathematical optimization techniques and computational tools in addressing real-world logistics challenges, offering insights that can inform decision-making in supply chain management and packaging design.

7. Contributions:

Muddassir Asghar (23i-2577):

Contributed to breaking down the problem into steps, conducted research to get more information about the problem and find ways to write the solution in a programming language and assisted in the coding by providing a prototype for the final code. Assisted Ibrahim in Quality Assurance of the model and brainstormed the step-by-step process of the model.

Ibrahim Kiani (23i-2536):

Provided the steps to solve the problem, created the objective function and constraint function. Assured quality of the model by manually testing inputs and checking the results. Provided guidance on how to solve the problem and features that need implementation.

Abdullah Ali (23i-2523):

Created the final python program to solve the problem, optimizing the code, and plotting the required graphs via python. Contributed to updating the project report and ensuring that the report is following instructions. Conducted research on how to use python to solve Lagrange Multiplier problems.

Difficulties faced during the assignment included understanding the Lagrange Multipliers method and implementing it in Python. These challenges were overcome through group discussions and online resources.

This project demonstrates the importance of mathematical optimization techniques in solving real-world logistics problems and highlights the potential for further research and refinement of the model to address additional constraints and variables.