Ejemplo Almuerzo

1. Valor calórico: 900 calorías (para un almuerzo)
2. El objetivo es minimizar las grasas totales del almuerzo.
3. Formula calórica:

macronutriente	Porcentaje	Kcal	gramos
Hidrato de Carbono	60%	900*0.6=540	540/4=135gr
Proteínas	15%	900*0.15=135	135/4=33.75gr
Grasas	25%	900*0.25=225	225/4=56.25

4. Un menú de un almuerzo podria contener menos de 900 calorias, por ejemplo en la siguiente tabla un menu especifico tendria un total de 357.34kcal:

alimento	Cant(grs.)	HdC(grs.)	Proteinas(grs.)	Grasas(grs.)	kcal
arroz	60	46.98	4.2	0.42	208.5
carne	33	0	6.11	5.64	75.18
cebolla	27	1.57	0.24	0.05	7.72
lenteja	20	11.3	4.78	0.18	65.94
		59.85	15.33	6.29	357.34

Donde kcal = hdc*4+proteinas*4+grasas*4

5. La tabla anterior se obtuvo a partir de los valores nutricionales por cada 100 gramos de la siguinte tabla de los nutreintes de un almuerzo:

alimento	HdC(grs.)	Proteinas(grs.)	Grasas(grs.)
arroz	78.3	7	0.7
carne	0	18.5	17.09
cebolla	5.8	0.9	0.2
lenteja	56.5	23.9	0.9

El Valor Nutricional es:

Valor Nutricional = (Cant. Utilizada (grs.) x Valor cada 100 grs.)/100

Ejemplo:

Según la Tabla anterior, **100 g de arroz** aportan **78.3 g de HdC.** En la penultima tabla, se usaron **60 g de arroz**.

Aplicamos la fórmula:

HdC = 60 x 78.3 / 100 = 46.98 grs

Este valor coincide con el de la **penultima tabla**.

6. Modelo matemático

En este punto se debe definir la funcion objetivo y las restricciones que responden a ña forma:

F.O.

MIN Z=c1.x1+c2.x2+ ...+ cn.xn)

Sujeto a:

a11.x1+a12.x2+...+a1n.xn<=b1

a21.x1+a22.x2+...+a1n.xn<=b2

.

.

Restricciones de no negatividad:

X1, x2, ..., xn >=0

Suponiendo que un profesional recomienda que un almuerzo debe contener:

30 grs de Proteina

130 grs de HdC

20 grs de Grasa

No mas de 60 grs de Carne

No mas de 100 grs de Verdura

No mas de 70 grs de Cereales y Legumbres

Podemos modelar las variables de desicion del siguiente modo:

CAR: Carne (en grs)

VER: Verdura (en grs)

CYL: Cereales y Legumbre (en grs)

Si el objetivo es minimizar la cantidad de grasas totales z entonces la funcion objetivo (usando la tabla del item 5) sería:

MIN z=grasas de carnes+grasas de verduras+grasas de cereales y legumbres

MIN z = 17.09*CAR+0.2*VER+(0.7+0.9)*CYL

Las restriciones de macronutrientes serian:

Proteinas >= 30 grs

Proteinas = Proteinas Carne + Proteinas Verdura + Proteinas Cereales y Legumbres

18.5*CAR+0.9*VER+(7+23.9)*CYL>=30

HdC>=130 grs

HdC = Hdc Carne + HdC Verdura + HdC Cereales y Legumbres

0*CAR+5.8*VER+(78.3+56.5)*CYL>=130

Restricciones maximas:

CAR<=60

VER<=100

CYL<=70

Restricciones de no negatividad:

CAR, VER, CYL >=0

7. Dado que el modelo matematico del almuerzo ahora esta planteado solo resta aplicar alguna tecnica de optimizacion metaheuristica como por ejemplo PSO o ABC (Artificial Bee Colony) pero que contemple algun mecanismo de manejo de las restricciones como por ejemplo la propuesta por Deb*.

8. Codigo fuente

```
import numpy as np
import random
import math

# --- Problem Definition ---
def objective_function(solution):
    """Calculates the objective function z to minimize."""
    car, ver, cyl = solution
    # Combine CYL coefficients: 0.7 + 0.9 = 1.6
    return 17.09 * car + 0.2 * ver + 1.6 * cyl

def calculate_violation(solution):
    """Calculates the total constraint violation for a solution."""
    car, ver, cyl = solution
    total_violation = 0.0
```

```
    # Constraint 1: 18.5*CAR + 0.9*VER + (7+23.9)*CYL >= 30
    # -> 30 - (18.5*CAR + 0.9*VER + 30.9*CYL) <= 0
    g1 = 30 - (18.5 * car + 0.9 * ver + 30.9 * cyl)
    total_violation += max(0, g1)

    # Constraint 2: 0*CAR + 5.8*VER + (78.3+56.5)*CYL >= 130
    # -> 130 - (5.8*VER + 134.8*CYL) <= 0
    g2 = 130 - (5.8 * ver + 134.8 * cyl)
    total_violation += max(0, g2)

    # Constraint 3: CAR <= 60 -> CAR - 60 <= 0
    h1 = car - 60
    total_violation += max(0, h1)

    # Constraint 4: VER <= 100 -> VER - 100 <= 0
    h2 = ver - 100
    total_violation += max(0, h2)

    # Constraint 5: CYL <= 70 -> CYL - 70 <= 0
    h3 = cyl - 70
    total_violation += max(0, h3)

    # Non-negativity is handled by bounds, but we can add explicit checks if needed
    # violation += max(0, -car)
    # violation += max(0, -ver)
    # violation += max(0, -cyl)

    return total_violation

def is_better(sol1_obj, sol1_viol, sol2_obj, sol2_viol):
    """
    Implements Deb's rules for comparing two solutions in minimization.
    Returns True if solution 1 is better than solution 2.
    """
    # Rule 1: Feasible is better than infeasible
    if sol1_viol == 0 and sol2_viol > 0:
        return True
    elif sol1_viol > 0 and sol2_viol == 0:
        return False
    # Rule 2: Less violation is better (if both infeasible)
    elif sol1_viol > 0 and sol2_viol > 0:
        return sol1_viol < sol2_viol
    # Rule 3: Better objective is better (if both feasible)
    else: # Both feasible (violation == 0)
        return sol1_obj < sol2_obj

# --- ABC Algorithm ---
def constrained_abc(objective_func, violation_func, bounds, num_food_sources, max_iterations, limit):
    """
    ABC algorithm for constrained optimization using Deb's rules.
```

```
Args:
    objective_func: Function to calculate objective value.
    violation_func: Function to calculate total constraint violation.
    bounds (list of tuples): Lower and upper bounds for each variable [(min1, max1), ...].
    num_food_sources (int): Number of food sources (half the colony size).
    max_iterations (int): Maximum number of cycles.
    limit (int): Threshold for scout phase activation.

Returns:
    tuple: Best feasible solution found, its objective value, its violation (should be 0 or close).
           Returns (None, float('inf'), float('inf')) if no feasible solution is found.
"""
dim = len(bounds)
lower_bounds = np.array([b[0] for b in bounds])
upper_bounds = np.array([b[1] for b in bounds])

# Ensure colony size is reasonable
colony_size = num_food_sources * 2
print(f"Initializing Colony: {num_food_sources} Employed, {num_food_sources} Onlookers")

# 1. Initialization
food_sources = np.zeros((num_food_sources, dim))
obj_values = np.full(num_food_sources, float('inf'))
violations = np.full(num_food_sources, float('inf'))
trial_counters = np.zeros(num_food_sources, dtype=int)

global_best_solution = None
global_best_obj = float('inf')
global_best_violation = float('inf')

print("Initializing Population...")
for i in range(num_food_sources):
    food_sources[i] = lower_bounds + np.random.rand(dim) * (upper_bounds - lower_bounds)
    obj_values[i] = objective_func(food_sources[i])
    violations[i] = violation_func(food_sources[i])

    # Update initial global best using Deb's rules
    if global_best_solution is None or \
       is_better(obj_values[i], violations[i], global_best_obj, global_best_violation):
         global_best_solution = food_sources[i].copy()
         global_best_obj = obj_values[i]
         global_best_violation = violations[i]

print(f"Initial Best Guess: Obj={global_best_obj:.4f}, Viol={global_best_violation:.4f}")

# --- Main Loop ---
for iteration in range(max_iterations):
    # 2. Employed Bee Phase
    for i in range(num_food_sources):
        # Select a different source k
        k = i
        while k == i:
```

```
            k = random.randint(0, num_food_sources - 1)

        # Select a dimension j to modify
        j = random.randint(0, dim - 1)

        # Generate neighbour solution
        phi = random.uniform(-1, 1)
        new_solution = food_sources[i].copy()
        new_solution[j] = food_sources[i, j] + phi * (food_sources[i, j] - food_sources[k, j])

        # Clamp to bounds
        new_solution = np.clip(new_solution, lower_bounds, upper_bounds)

        # Evaluate the new solution
        new_obj = objective_func(new_solution)
        new_violation = violation_func(new_solution)

        # Apply Deb's rules for replacement
        if is_better(new_obj, new_violation, obj_values[i], violations[i]):
            food_sources[i] = new_solution
            obj_values[i] = new_obj
            violations[i] = new_violation
            trial_counters[i] = 0
        else:
            trial_counters[i] += 1

    # 3. Onlooker Bee Phase
    # Calculate probabilities - Use a simple fitness proxy for selection
    # Higher fitness for lower objective (or lower violation if infeasible)
    # Avoid division by zero or issues with large values.
    fitness_proxy = np.zeros(num_food_sources)
    for i in range(num_food_sources):
         # Prefer feasible solutions, then better objectives, then lower violations
         if violations[i] == 0:
             # Use inverse of objective for feasible (add small epsilon)
              fitness_proxy[i] = 1.0 / (1.0 + obj_values[i] + 1e-6)
         else:
             # Use inverse of violation for infeasible (lower violation = higher fitness)
             fitness_proxy[i] = 1.0 / (1.0 + violations[i] + 1e-6)



    total_fitness = np.sum(fitness_proxy)
    if total_fitness <= 0: # Fallback if all fitness is zero
        probabilities = np.ones(num_food_sources) / num_food_sources
    else:
        probabilities = fitness_proxy / total_fitness
        # Ensure probabilities sum to 1 due to potential floating point issues
        probabilities /= np.sum(probabilities)

    for onlooker in range(num_food_sources):
        try:
            # Ensure no NaN or negative probabilities
```

```
                if np.any(np.isnan(probabilities)) or np.any(probabilities < 0):
                     # Fallback: uniform probability if calculation failed
                     selected_index = np.random.choice(range(num_food_sources))
                else:
                     selected_index = np.random.choice(range(num_food_sources), p=probabilities)
            except ValueError: # Handles cases where probabilities don't sum to 1 exactly
                # Fallback: uniform probability
                selected_index = np.random.choice(range(num_food_sources))


            # Generate neighbour solution for the selected source
            k = selected_index
            while k == selected_index:
                k = random.randint(0, num_food_sources - 1)
            j = random.randint(0, dim - 1)
            phi = random.uniform(-1, 1)

            new_solution = food_sources[selected_index].copy()
            new_solution[j] = food_sources[selected_index, j] + phi * (food_sources[selected_index, j] -
food_sources[k, j])
            new_solution = np.clip(new_solution, lower_bounds, upper_bounds)

            new_obj = objective_func(new_solution)
            new_violation = violation_func(new_solution)

            # Apply Deb's rules for replacement for the selected source
            if is_better(new_obj, new_violation, obj_values[selected_index], violations[selected_index]):
                food_sources[selected_index] = new_solution
                obj_values[selected_index] = new_obj
                violations[selected_index] = new_violation
                trial_counters[selected_index] = 0
            else:
                trial_counters[selected_index] += 1

        # Update Global Best after Employed and Onlooker phases
        for i in range(num_food_sources):
             if is_better(obj_values[i], violations[i], global_best_obj, global_best_violation):
                 global_best_solution = food_sources[i].copy()
                 global_best_obj = obj_values[i]
                 global_best_violation = violations[i]


        # 4. Scout Bee Phase
        for i in range(num_food_sources):
            if trial_counters[i] > limit:
                # Replace with a new random solution
                food_sources[i] = lower_bounds + np.random.rand(dim) * (upper_bounds - lower_bounds)
                obj_values[i] = objective_func(food_sources[i])
                violations[i] = violation_func(food_sources[i])
                trial_counters[i] = 0

                # Check if the new random solution is the best so far
                if is_better(obj_values[i], violations[i], global_best_obj, global_best_violation):
```

```
                global_best_solution = food_sources[i].copy()
                global_best_obj = obj_values[i]
                global_best_violation = violations[i]

        # Optional: Print progress
        if (iteration + 1) % 50 == 0:
            print(f"Iter {iteration + 1}/{max_iterations}: Best Obj={global_best_obj:.4f}, 
Viol={global_best_violation:.4f}")

    print("\nOptimization Finished.")
    # Final check if a feasible solution was found
    if global_best_violation > 1e-6: # Use a small tolerance for feasibility
        print("Warning: No feasible solution found within the tolerance.")
        # Return the least infeasible solution found
        return global_best_solution, global_best_obj, global_best_violation

    return global_best_solution, global_best_obj, global_best_violation

# --- Parameters ---
problem_bounds = [
    (0, 60),    # Bounds for CAR
    (0, 100),   # Bounds for VER
    (0, 70)     # Bounds for CYL
]
num_sources = 30      # Number of employed/onlooker bees
iterations = 500      # Number of algorithm cycles
abandonment_limit = 40 # Limit for trial counter before scout phase

# --- Run ABC ---
best_solution, best_objective, best_violation = constrained_abc(
    objective_function,
    calculate_violation,
    problem_bounds,
    num_sources,
    iterations,
    abandonment_limit
)

# --- Print Results ---
if best_solution is not None:
    print("\n--- Optimal Solution Found ---")
    print(f"Variables (CAR, VER, CYL): {np.round(best_solution, 4)}")
    print(f"Optimal Objective Value (z): {best_objective:.4f}")
    print(f"Constraint Violation: {best_violation:.6f}") # Should be close to 0

    # Verify constraints with the found solution
    print("\nConstraint Verification:")
    car_s, ver_s, cyl_s = best_solution
    c1_val = 18.5*car_s + 0.9*ver_s + 30.9*cyl_s
    c2_val = 5.8*ver_s + 134.8*cyl_s
    print(f"Constraint 1 (>= 30): {c1_val:.4f} -> {'Met' if c1_val >= 30 - 1e-6 else 'VIOLATED'}")
    print(f"Constraint 2 (>= 130): {c2_val:.4f} -> {'Met' if c2_val >= 130 - 1e-6 else 'VIOLATED'}")
    print(f"Constraint 3 (CAR <= 60): {car_s:.4f} -> {'Met' if car_s <= 60 + 1e-6 else 'VIOLATED'}")
```

```
    print(f"Constraint 4 (VER <= 100): {ver_s:.4f} -> {'Met' if ver_s <= 100 + 1e-6 else 'VIOLATED'}")
    print(f"Constraint 5 (CYL <= 70): {cyl_s:.4f} -> {'Met' if cyl_s <= 70 + 1e-6 else 'VIOLATED'}")
else:
    print("\nCould not find a feasible solution.")
```

* Kalyanmoy Deb, “An efficient constraint handling method for genetic algorithms”, *Computer Methods in Applied Mechanics and Engineering*, vol. 186, pp. 311-338, 2000.