## **Problem two Manufacturing Problem**

## **Input Parameter Generation**

**N = np.random.randint(1,11)**

**M = np.random.randint(1,11)**

**W = np.random.randint(1,11)**

**Pn = [round(random.uniform(10, 30), 2) for \_ in range(20)]**

**Dnm = np.random.randint(0, 2, size=(N, M))**

**Swm = np.random.randint(0, 2, size=(W, M))**

**Tnm = np.round(np.random.uniform(0.01, 1, size=(N, M)), 2)**

**MaxMt = 8**

**MaxWt = 8**

**M\_max = 1000**

*In this section we randomly initialize the problem parameters:*

* ***N, M, W:*** *Number of products, machines, and workers (randomly between 1 and 10).*
* ***Pn:*** *Profit per unit for each product (random float between 10 and 30).*
* ***Dnm:*** *Binary matrix showing which products can be produced on which machines.*
* ***Swm:*** *Binary matrix showing which workers can operate which machines.*
* ***Tnm:*** *Time required to produce one unit of product n on machine m (float between 0.01 and 1).*
* ***MaxMt, MaxWt:*** *Daily time limit per machine and per worker (8 hours).*
* ***M\_max:*** *A large number used to set an upper bound for product n on machine m*

## 

## **2. Objective Function**

**def obj\_fn(P, Y):**

**profit = 0**

**for n in range(N):**

**for m in range(M):**

**profit += P[n] \* Y[n][m]**

**return profit**

*Calculates the total profit of a production plan by summing up:*

*profit = ∑ P[n] × Y[n][m]  
 Where Y[n][m] is the number of units of product n produced on machine m.*

## **3. Constraint Checking Function**

**def check\_constraints(Y, Z, T, D, S):**

*# 1. Product–machine compatibility:*

*# A product n can only be assigned to a machine m if D[n][m] =1*

*# M\_max is a large constant used to enforce this as a soft constraint*

**for n in range(N):**

**for m in range(M):**

**if Y[n][m] > M\_max \* D[n][m]:**

**return False, "Product–machine compatibility"**

*# 2. Machine-time capacity:*

*# Each machine m can operate at most 8 hours total across all products.*

**for m in range(M):**

**machine\_time = sum(T[n][m] \* Y[n][m] for n in range(N))**

**if machine\_time > 8:**

**return False, "Machine-time capacity"**

*# 3. Worker-time capacity:*

*# Each worker w can work no more than 8 hours total across all machines.*

**for w in range(W):**

**worker\_time = sum(Z[w][m] for m in range(M))**

**if worker\_time > 8:**

**return False, "Worker-time capacity"**

*# 4. Worker–machine compatibility:*

*# A worker w can only be assigned time on machine m if they are skilled for it (S[w][m] = 1)*

*# Otherwise, Z[w][m] should be zero.*

**for w in range(W):**

**for m in range(M):**

**if Z[w][m] > 8 \* S[w][m]:**

**return False, "Worker–machine compatibility"**

*# 5. Covering production time by workers:*

*# The total time all workers spend on machine m must equal the machine's total required runtime.*

**for m in range(M):**

**left = sum(Z[w][m] for w in range(W)) #Total worker time assigned to machine m**

**right = sum(T[n][m] \* Y[n][m] for n in range(N)) #Total runtime machine m needs**

**if not np.isclose(left, right, atol=1e-2):**

**return False, "Covering production time by workers"**

*# 6. Non-negativity and type constraints:*

*# Y[n][m] must be a non-negative integer (number of product units assigned to machine)*

**for n in range(N):**

**for m in range(M):**

**if Y[n][m] < 0 or not np.issubdtype(type(Y[n][m]), np.integer):**

**return False, "Non-negativity & type (Y)"**

*# Z[w][m] must be non-negative (time worker spends on machine)*

**for w in range(W):**

**for m in range(M):**

**if Z[w][m] < 0:**

**return False, "Non-negativity (Z)"**

**return True, "good"**

## **4. Feasible Agent Generation**

**def generate\_feasible\_agent():**

**...**

**return y, z**

*The aim of this function is to build a feasible initial solution (agent) by:*

* *Iterating over machines.*
* *Identifying valid products and eligible workers.*
* *Randomly assigning producible units of valid products to machines.*
* *Distributing the required production time across available workers.*

*If allocation fails, it rolls back the assignment.*

*Ensures all initial agents satisfy constraints before being used in PSO.*

**5. Initial Agent Pool Creation**

**agents\_y = []**

**agents\_z = []**

**personal\_best\_scores = []**

**while len(agents\_y) < num\_agents:**

**y, z = generate\_feasible\_agent()**

**test, msg = check\_constraints(y, z, Tnm, Dnm, Swm)**

**print(msg)**

**if test:**

**agents\_y.append(y)**

**agents\_z.append(z)**

**objective = obj\_fn(Pn, y)**

**personal\_best\_scores.append(objective)**

### *In this part we initialize a pool of 10 feasible agents.*

### *Each agent has: 1- y: Unit production plan.*

### *2- z: Worker time allocation.*

### *3- Keep track of their personal best profit scores.*

*Only agents that pass all constraints are accepted.*

## **6. Initializing PSO Components**

**velocities\_Y = [np.zeros((N, M)) for \_ in range(num\_agents)]**

**velocities\_Z = [np.zeros((W, M)) for \_ in range(num\_agents)]**

**personal\_best\_Y = agents\_y.copy()**

**personal\_best\_Z = agents\_z.copy()**

**g\_best\_index = np.argmax(personal\_best\_scores)**

**global\_best\_Y = personal\_best\_Y[g\_best\_index].copy()**

**global\_best\_Z = personal\_best\_Z[g\_best\_index].copy()**

### *Purpose:*

### *Initializes velocities of each agent to zero.*

### *Saves each agent’s initial solution as their personal best.*

### *Determines the global best solution based on maximum profit so far.*

## **7. Main PSO Loop**

**for iter in range(iterations):**

**for i in range(num\_agents):**

**...**

### *Purpose: Runs the Particle Swarm Optimization for a specified number of iterations (e.g., 100). Each iteration includes:*

### *a. Velocity Update: Combines inertia, personal experience, and social influence to adjust Y and Z velocities.*

**velocities\_Y[i] = (w \* velocities\_Y[i] +**

**c1 \* r1 \* (personal\_best\_Y[i] - agents\_y[i]) +**

**c2 \* r2 \* (global\_best\_Y - agents\_y[i]))**

### *b. Position Update: Updates each agent's production and worker allocation plans.*

**temp\_y = np.round(agents\_y[i] + velocities\_Y[i]).astype(int)**

**temp\_z = agents\_z[i] + velocities\_Z[i]**

#### *c. Constraint Check: Invalid updates are discarded.*

**cond, a = check\_constraints(temp\_y, temp\_z, Tnm, Dnm, Swm)**

**if not cond:**

**continue**

#### *d. Evaluate and Update Best: If a new solution is better, update personal and global bests accordingly.*

**score = obj\_fn(Pn, agents\_y[i])**

**if score > personal\_best\_scores[i]:**

**...**

## **8. Final Results Display**

**print("\n=== Final Best Solution ===")**

**print("Best Profit:", personal\_best\_scores[g\_best\_index])**

**print("Y (Units Produced):")**

**print(global\_best\_Y)**

**print("Z (Worker Time Allocation):")**

**print(np.round(global\_best\_Z, 2))**

*After all iterations:*

* *Displays the best profit found.*
* *Shows the optimal production plan (Y) and worker allocation (Z).*