Particle swarm optimization (PSO):

Particle Swarm Optimization [Kennedy and Eberhart, 1995] is a population based heuristic optimization algorithm inspired by social behavior of birds flocking or fish schooling.

Consider the scenario of birds looking for food. The birds first search their own neighborhood for the food source. At the end of each time step or iteration the birds decide on a location that might lead to a food source.

The birds then compare their solution with other birds' solutions and move closer to the birds that are closest to the location of the food source.

Each bird (a particle) is a potential solution in the search space. This concept is formulated as PSO algorithm.

Each particle is treated as a point in a D-dimensional space. Initially, N particles are uniformly distributed in the solution space. The particles in PSO fly through the search space with a certain velocity, and change their position dynamically in the hope of reaching the food source, the destination. Therefore, position and velocity are two important parameters in the PSO algorithm.

Each particle keeps track of the best position it has encountered during its travel, and the best position traveled by the swarm of particles. The best position traveled by a particle is called the local best position, and the best position traveled by the swarm is called the global best position. At the end of each iteration, the particles calculate their next velocity, and update their positions based on the calculated velocity.

PSO Algorithm

Particle ***i*** is represented as ***Xi =(xi1,xi2,…..,XiD***) , which represents a potential solution to a problem in D-dimensional space.

Each particle keeps a memory of its previous best position, ***Pbest***, and a velocity along each dimension, represented as ***Vi=(vi1,vi2,….,viD).***

For each iteration, the position of the particle with the best fitness value in the search space, designated as ***g***, and the ***P*** vector of the current particle are combined to adjust the velocity along each dimension, and that velocity is then used to compute a new position for the particle.

The method could be divided into GBEST and LBEST versions, whose main difference is their definition of the best. In the GBEST version, the particle swarm optimizer keeps track of the overall best value, and its location, obtaining thus far by any particle in the population, which is called ***gbest* (*Pgd*)**.

For the LBEST version, in addition to *gbest*, each particle keeps track of the best solution, called lbest(Pgd) , and it is attained within a local topological neighborhood of particles. However, the particle velocities in each dimension are held to a maximal velocity, ***Vmax***, and the velocity in that dimension is limited to *Vmax*.

Later, inertial weight was developed to better balance exploration and exploitation in order to eliminate the need for *Vmax*.

The updating rule is given by

- *Vid(new)= W.Vid(old) + c1.rand1.(Pid + Xid) + c2.rand2.(Pgd – Xid).*
- *Xid(new)= Xid(old) + Vid(new).*

Where ***W*** is the inertia weight, ***c1*** and ***c2*** determine the relative influence of the social and cognition components (learning factors), while ***rand1*** and ***rand2*** denote two random numbers uniformly distributed in the interval [0, 1]. Recent work done by Clerc indicates that the use of a *constriction factor* may be necessary to ensure convergence of the PSO algorithm . PSO algorithm has been applied for many optimization problems, such as scheduling and traveling sales problems.

Particle swarm optimization & Linear Programing Methodology:

The PSO concept originated as a simulation of a simplified social system. Each particle keeps track of its coordinates in the problem space which are associated with the best solution (fitness) it has achieved so far. Due to this merit, this study employs PSO algorithm for solving the LPPs(Linear Programing Problems).

The LPP is developed for decentralized planning systems in which the upper level is termed as the leader and the lower level pertains to the objective of the follower. In the LPP, each decision maker tries to optimize its own objective function without considering the objective of the other party, but the decision of each party affects the objective value of the other party as well as the decision space. This section will present the way to apply PSO algorithm for LPPs. The PSO algorithm for solving LPPs is presented as follows:

Step 1:

- Set up parameters including population size (the number of particles), Maximal velocity ***Vmax* ,** inertial weight ***W***, and two learning factors, (**c1** and **c2**). Two random variables, ***rand1*** and ***rand2*** , are in the interval [0, 1].

Step 2:

- Set up the searching range for x, which will influence the searching speed. Initialize each particle randomly with initial position, Xid, within the pre-specified range and velocity, Vid , in the range of maximal speed, Vmax. This study adopts the float coding method to generate the random numbers for the upper-level variables. Thus, every particle represents the real dimensional position. Then, program for variable ys in the lower level. Each particle's position is represented as:
  ***Xid* = (*xi1*,…..,*xin*,*yi1*,……,*yim*).**

Step 3 :

- Calculate every particle's fitness value using
- F = *CXid*.

Step 4:

- Update the local best position, ***Pid*** , and global best position, ***Pgd* .**

Step 5:

- According to the updated ***Pid*** and ***Pgd***, use this Eqs. to update the velocity and position for every particle.
- *Vid(new)= W.Vid(old) + c1.rand1.(Pid + Xid) + c2.rand2.(Pgd – Xid).*
- *Xid(new)= Xid(old) + Vid(new).*
- Every particle's velocity is constrained by the pre-determined maximal velocity ,***Vmax*** , and every particle's position,***Xid*** , should be within the specified range: l <= xi <=u .

Step 6:

- Stop if the specified number of generations is satisfied; otherwise, go back to Step 3.

Particle swarm optimization for option Pricing:

One of the important research problems in computational finance is identifying the best time to exercise a given option while maximizing the profit of the option.

The focus of this thesis is to design an algorithm that is inspired from natural world such as school of fish, flock of birds or in general swarm.

This algorithm is expected to compute optimal option values in the given solution space at a reasonable amount of time which is better than we consider a popular nature inspired algorithm, Particle Swarm Optimization (PSO). The first focus of our research is to design and develop a sequential algorithm for the option pricing problem using basic principles of PSO.

Mapping PSO for Option Pricing:

In mapping the PSO to the option pricing problem, a particle in our algorithm

is defined by five parameters: id, position vector ($\vec{X}$), velocity vector ($\vec{V}$), local best

position ($\vec{lbest}$) and global best position ($\vec{g\ best}$).$\vec{}$ ***X*** corresponds to the price and$\vec{}$ ***V***

corresponds to the change in price. ***Xmax*** and ***Xmin*** correspond to the boundaries of

the solution space. Similarly, previous best position ***P i(t)*** in PSO is a particle's best ***li*** best) in terms of option value (profit) and option exercise time; and ***Pg(t)*** in the PSO is the overall best (***g best***) among particles in the option pricing solution space.

initial particle position and velocity are already defined in Equations.

A particle is used to compute option value and capture the time when this optimum option value occurs. Therefore, each position vector ($\vec{X}$) has two dimensions.

The first dimension ***(X[0])*** is for stock price, and the second dimension (***X[1]***) is the

exercise time. Similarly, there are two dimensions for$\vec{}$ V ,$\vec{}$ ***lbest*** and$\vec{}$***g best***, ***Xmax*** and ***Xmin***.

Initially, the particles are distributed randomly in the solution space using initial postion Equation.

The initial velocity for the ***ith*** particle is calculated using his equation. The particles fly through

the search space with a certain velocity, and change their position dynamically until the termination condition is satisfied. In the current implementation the termination condition is set to a predetermined number of iterations.

The position and velocity updates by a particle i are calculated as follows:

$$Xi\ t+1[k] = Xi\ t\ [k] + Vi\ t+1[k]; \quad k = 0, 1.$$

$$Vi\ t+1[k] = \omega \times Vi\ t\ [k] + c1 \times r1 \times (li\ best[k] - Xi\ t\ [k]) + c2 \times r2 \times (gbest[k] - Xi\ t\ [k]); \quad k = 0, 1.$$

There are four possible scenarios to consider for a particle to update its li best[k] and gbest[k] after it moved to a new position. Figures 1- 3 demonstrate these scenarios for a call option. In these figures, "X" represents local best position, and a circle represents a new position of a particle.

Scenario I :

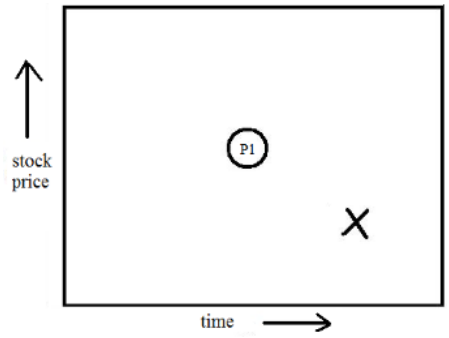


Figure 1: A particle movement for a call option Scenario I

A particle flies to a new position P1 (node), and checks if that new position represents better profit (higher local pay-off; that is, higher stock price for a call option) and represents an earlier time than the local best (price and time) obtained so far for that particle (see Figure 1).

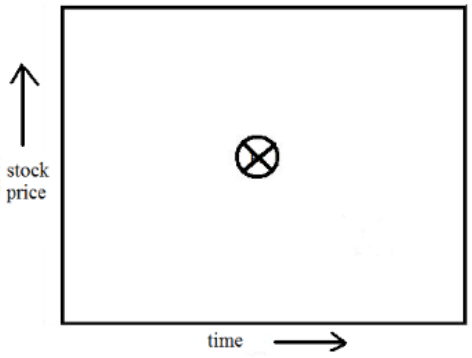


Figure 2: Local best update

If so, the particle updates its local best position as shown in Figure 2. Recall that we are demonstrating the particles movement for a call option. Similar arguments can be extended to

a put option.

Scenario II :

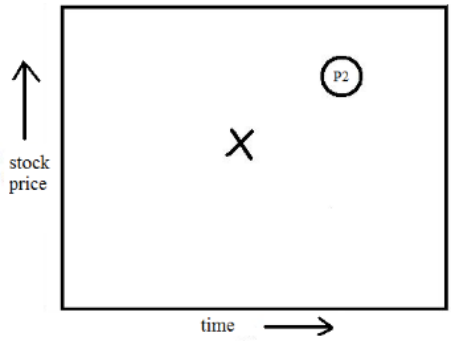


Figure 3: A particle movement for a call option Scenario II

A particle flies to a new position P2, and checks if that new position represents

better profit (higher stock price for a call option) and represents a later (future)

time than the local best position (price and time) obtained so far (see Figure 3)

for that particle. If so, the particle does the following:

- the option price at the new position (future time) is discounted using ***Xi[0] − K) × exp(−r × (Xi[1] – li best[1])),*** (where K is the strike price and is the risk free interest rate), to find its value at the local best position. Recall that ***X[1]*** and ***lbest[1]*** correspond to time.
- if this discounted value is greater than the current local best option price, then the particle flies to the new position (that is, waiting for the future time is beneficial), and updates its local best position with the newly computed option price and exercise time. Otherwise, the local best remains as it is.

Scenario III :

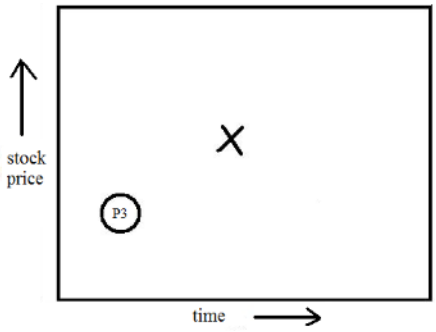


Figure 4: A particle movement for a call option Scenario III

A particle flies to a new position P3, and in that position if profit is less (i.e.lower local pay-off; that is lower stock price for a call option) and time in this new position represents an earlier time, then the particle does the following. Though the local pay-off is less at this new position, the time represented by this position is an earlier time than the local best time (see Figure 4). There

is a chance that the newly found node may move to profitable position (with higher local pay-off) at later time (iterations) than the local best position. In this case, the particle needs to search the surrounding neighborhood. This is to prevent stagnation at one location. The particle does the following:

- the future option value is discounted using ***(Xi[0]−K)×exp(−r×(libest[1]−Xi[1]))*** to calculate the current option value.
- if the discounted future option value is greater than the current local best option price, then particle updates its local best position. Otherwise, the local best remains as it .

Scenario IV :

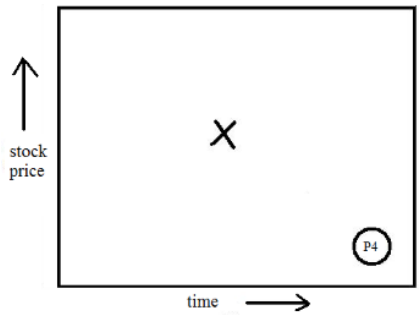


Figure 5: A particle movement for a call option Scenario IV

The last case where there is less profit (i.e. smaller local pay-off) at a later date (see Figure 5) in time, and is not advantageous because (a) particles may have already found a better position which is profitable and corresponds to an earlier time (b) particles are close to maturity date. In this case, a particle performs no update on its local best position.

After each iteration, the best of all the local best ($\vec{}$***li best***) is calculated in the same was as the above four cases in comparison with$\vec{}$***g best***. If ***(*** $\vec{}$ ***li best)*** is better than current$\vec{}$***g best*** (global best), then$\vec{}$***g best*** will be updated. The update in local and global best positions continues until termination condition is met.

PSO-based algorithm for Option Pricing:

The pseudo code for a sequential PSO-based option pricing algorithm is given below:

Initialize PSO parameters: ***c1, c2, ω,→ Vi max,→ Vi min,→ Xi max,→ Xi min***

Initialize option pricing variables and parameters: S, K, r, σ, T

for i = 1 to N do

Initialize→ ***Xi*** by using Equation ***Xi0 = Xmin + r1 × (Xmax - Xmin)***

Initialize→ ***Vi*** by using Equation ***Vi0 = Xi***

Li best =→ ***Xi***

end for→

g best = max→ l1best,→ l2best, . . . ,→ lNbest

for ***Itr*** = 1 to M AXIT ER do

for i = 1 to N do

Calculate the new position (→***Xi I(tr))*** and velocity (→***Vi (Itr***)) by using Equations

Xit+1[k] = Xit [k] + V it+1[k]; k = 0, 1.

and Vi t+1[k] = ω×Vi t [k]+c1×r1×(li best[k]Xi t [k])+c2×r2×(gbest[k]−Xi t [k]); k = 0, 1. respectively.

Check boundary conditions: If a particle flies out of boundary (solution space i.e.→ ***Xi max*** or→ ***Xi min***), then re-calculate new velocity, and check boundary conditions one more time. If a particle flies out of solution space with re-calculated

velocity, then penalize the particle with some random velocity.

Update the local best position by considering the four scenarios discussed in above .

end for→

g best = max→ l1best,→ l2best, . . . ,→ lN best

Update→***g best*** information in all particles

end for

Prin t→***g best***

1- Unit price Proposal problem:

Bidwell Construction is faced with the task of preparing a bid for excavation work on a unit price proposal. This endeavor involves determining competitive unit prices for various excavation tasks while ensuring profitability. The company must navigate through cost calculations, markup considerations, and market dynamics to formulate a bid that not only covers expenses but also generates a desirable profit margin. This scenario presents a classic optimization problem in the realm of construction project bidding, where linear programming techniques can be employed to find the most advantageous solution.

2-Problem Overview:

The problem at hand involves several key components:

- Identifying the unit costs for different excavation tasks such as clearing, earth excavation, rock excavation, and cleanup.
- Applying a 5% markup to these unit costs to establish unit prices for the bid.
- Ensuring that the total bid amount, incorporating all unit prices, aligns with the desired budget and profit margin.
- Balancing the unit prices for rock excavation and earth excavation to maintain competitiveness in the bidding process.
- Expressing all financial transactions in terms of present worth to facilitate effective decision-making.

3-Objectives:

To solve this problem effectively, Bidwell Construction must:

1. Determine optimal unit prices for each excavation task that minimize the total cost of the bid while meeting profitability requirements.
2. Ensure that the bid proposal remains competitive in the market by appropriately pricing each item of excavation work.
3. Utilize linear programming techniques to optimize the bidding strategy and navigate through various constraints.
4. Interpret the results to make informed decisions regarding the final bid proposal, considering both financial considerations and market dynamics.

By addressing these objectives systematically, Bidwell Construction can develop a compelling bid proposal that not only meets the client's needs but also enhances the company's profitability and competitiveness in the construction industry.

4-Problem Analysis:

Objective: The objective is to determine the unit prices (X1, X2, X3, X4) for different items in Bidwell Construction's unit price proposal in order to maximize the total bid value.

Constraints:

1. **Total Bid Value Constraint:** The total bid value should include a 5% markup, which should sum up to $262,500.
    - $20{,}000x1+75{,}000x2+25{,}000x3+20{,}000x4=262{,}500$
2. **Rock Excavation Unit Price Constraint:** The unit price for rock excavation should be higher than the unit price for earth excavation to ensure a balanced bid.
    - $x3-x2\geq 0$
3. **Unit Price Limits Constraint:** Bidwell has specified certain limits for the unit prices these constraints ensure that the unit prices fall within the specified rang.
    - $\$1.00 \leq x1 \leq \$4.00$
    - $\$0.50 \leq x2 \leq 3.00$
    - $\$1.50 \leq x3 \leq 6.00$
    - $\$1.00 \leq x4 \leq \$4.00$
4. **Non-Negativity Constraint:** The unit prices for all items should be non-negative.
    - $x1,x2,x3,x4\geq 0$

Decision Variables:

- x1: Unit price for Clearing
- x2: Unit price for Earth Excavation
- x3: Unit price for Rock Excavation
- x4: Unit price for Cleanup

Objective Function: The objective is to maximize the total bid value, which is the sum of the product of unit prices and quantities for each item.

- $Z=19{,}606x1+67{,}897x2+22{,}632x3+17{,}749x4$

The code implementation

The encoding

The positions and velocities of the particles in the search space are represented by vectors encoded in the PSO algorithm. A possible solution to the optimization problem correlates with the position of each particle. The values of the variables (x) in this scenario that require optimization are represented by the position vector while keeping the values of the decision variables in the domain.

```
# Initialize particle attributes with
self.position = np.random.rand(dim)
self.velocity = np.random.rand(dim)
```

```
x_max = [4 , 3 , 6 , 4]
x_min = [1 , 0.5 , 1.5 , 1]
```

The operators

The velocity of each particle is updated using the PSO equations. The new velocity is calculated based on the particle's current velocity to keep the direction without changing, the difference between the particle's current position and its best local position, and the difference between the particle's current position and the global best position. The cognitive coefficient and social coefficient determine the influence of the particle's best local position and the global best position, respectively.

```
def update_velocity(particle, particle_local_position , global_best_position,  cognitive_coeff, social_coeff):
    # Update particle velocity based on PSO equations
    new_velocity = (particle.velocity +
                    cognitive_coeff * np.random.rand(len(particle.position)) * (particle_local_position - particle.position) +
                    social_coeff * np.random.rand(len(particle.position)) * (global_best_position - particle.position))
    return new_velocity
```

The position of each particle is updated based on its current position and velocity. The new position is calculated by adding the velocity vector to the current position vector. Bounds are applied to ensure that the new position remains within the defined search space.

```
def update_position(particle):
    # Update particle position based on velocity and bounds
    new_position = particle.position + particle.velocity
    for i in range(len(new_position)):
        new_position[i] = np.minimum(new_position[i] ,x_max[i])
        new_position[i] = np.maximum(new_position[i] ,x_min[i])
    return new_position
```

The constraints

the constraints of the optimization problem are defined as functions, such as "total_bid_constraint" and "balance_constraint". These functions check whether a given particle's position satisfies the corresponding constraint. In the code, before evaluating the fitness of a particle, it is checked whether all constraints are satisfied for that particle's position.

```
def total_bid_constraint(x):
    # Total bid constraint function
    return 262500 - (20000 * x[0] + 75000 * x[1] + 25000 * x[2] + 20000 * x[3])

def balance_constraint(x):
    # Balance constraint function
    return x[2] - x[1]
```

The logical flow of the program

A Particle class with attributes for position, velocity, and fitness. The initial position and velocity of each particle are set to random values.
The optimization problem's objective function, balance constraint function, and total bid constraint function are all defined in the program.
The particle's velocity is updated based on the PSO equations using the social and cognitive coefficients, along with the particle's current position, local best position, and global best position.
The update_position function, which the program defines, changes the particle's position depending on its velocity and current position while making sure the new position stays inside the specified search space constraints.

the PSO function, which implements the Particle Swarm Optimization algorithm. It takes the objective function, constraints, and optional parameters such as the number of particles and maximum iterations.

Inside the PSO function, a list of particles is initialized with random positions and velocities.

Each particle's fitness is evaluated by applying the constraints and calculating the objective function value for its position.
If a particle's fitness is higher than its local best fitness, the local best fitness and position are updated for that particle.
If the local best fitness is higher than the global best fitness, the global best fitness and position are updated.
The global best fitness value at each iteration is stored in a list for tracking the convergence.
The PSO algorithm concludes by returning the global best position, global best fitness, and the history of best fitness values.
The program defines the constraints and bounds for the optimization problem.
The PSO function is called with the objective function and constraints, and the results (global best position, global best fitness, and best fitness history) are assigned to variables.
The program outputs the results or further processes them as needed.

The algorithm parameters

Number of Particles (num_particles = 100):
This parameter determines the number of particles in the swarm. It represents the population size and affects the exploration and exploitation capabilities of the algorithm.

Maximum Iterations (max_iterations = 500):
The maximum number of generations or iterations. It establishes the algorithm's stopping criterion. By increasing the maximum number of iterations, the algorithm might potentially improve the quality of the result by exploring the search space for a longer period of time.

Cognitive Coefficient (cognitive_coeff = 1.5):
The cognitive coefficient determines the influence of a particle's best local position on its velocity update. A higher cognitive coefficient gives more weight to the particle's historical best position, leading to more exploitation. Conversely, a lower cognitive coefficient promotes more exploration by giving less weight to the particle's historical information.

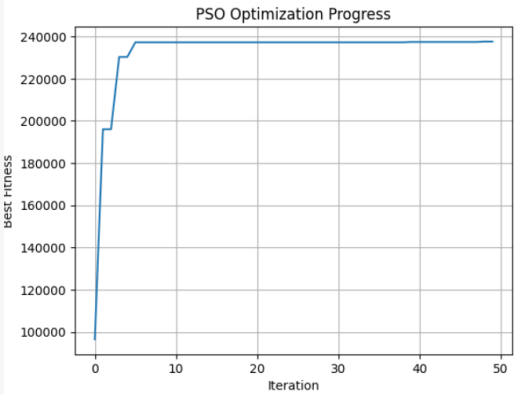
Social Coefficient (social_coeff = 2.5):
The social coefficient determines the influence of the global best position on a particle's velocity update. It represents the collective behavior of the swarm and encourages particles to move towards the best solution found globally. A higher social coefficient emphasizes exploitation by placing more importance on the global best position. A lower social coefficient promotes exploration by reducing the influence of the global best position.

Examples on the problem size

Using number of iterations (50) and number of particles (10)

```
# Print results
print("Optimal unit prices:")
print("Item 1:", best_position[0])
print("Item 2:", best_position[1])
print("Item 3:", best_position[2])
print("Item 4:", best_position[3])
print("Total cost:", best_fitness)
```

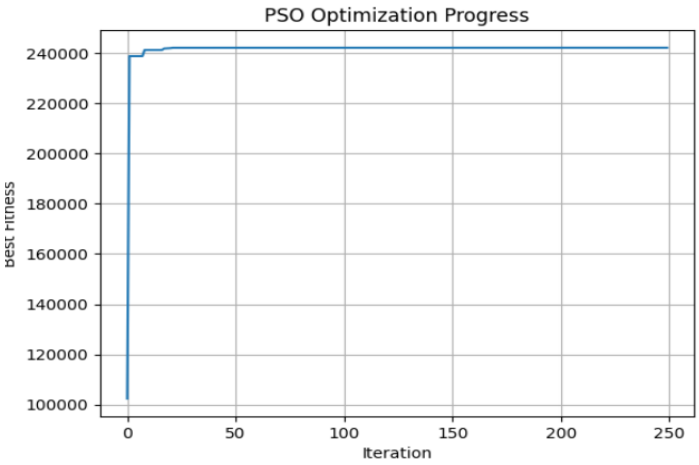
```
Optimal unit prices:
Item 1: 1.0
Item 2: 0.5
Item 3: 5.136615030725355
Item 4: 3.819647309417074
Total cost: 237601.2914702199
```



Using number of iterations (250) and number of particles (50)

```
# Print results
print("Optimal unit prices:")
print("Item 1:", best_position[0])
print("Item 2:", best_position[1])
print("Item 3:", best_position[2])
print("Item 4:", best_position[3])
print("Total cost:", best_fitness)
```

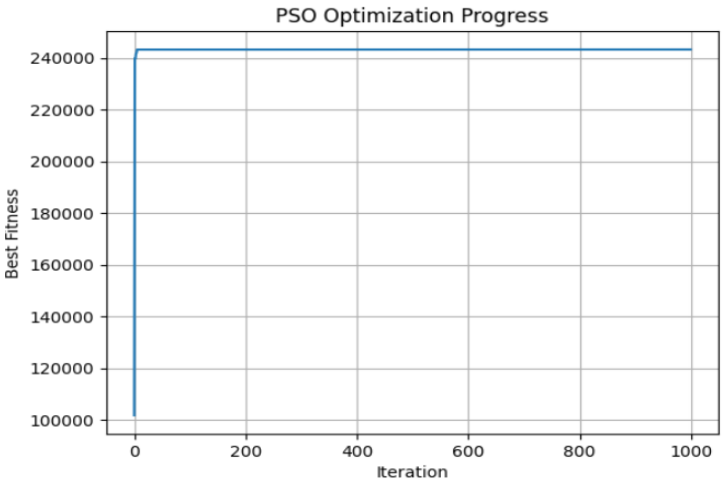
```
Optimal unit prices:
Item 1: 4.0
Item 2: 0.8666282614191617
Item 3: 1.5
Item 4: 4.0
Total cost: 242209.45906557684
```



Using number of iterations (1000) and number of particles (250)

```
# Print results
print("Optimal unit prices:")
print("Item 1:", best_position[0])
print("Item 2:", best_position[1])
print("Item 3:", best_position[2])
print("Item 4:", best_position[3])
print("Total cost:", best_fitness)
```

```
Optimal unit prices:
Item 1: 2.7499855239597073
Item 2: 0.5
Item 3: 6.0
Item 4: 1.0
Total cost: 241405.71618275403
```



Based on the results we need to take a moderate number of iterations and number of particles and perform a number of generations to determine the maximum profit that would be after implementing the program 243260.161$ with a price for the variables

Item 1: 4.0
Item 2: 1.138
Item 3: 3.082
Item 4: 1.0

Fuzzification

After running the algorithm and getting the optimum values for the decision variables we picked the x1 parameter to consider as a crisp value

And convert it to a membership value by fuzzifying it to 3 categories (low, medium, high) to convert it to a linguistic instead of just a number because of the precision.

```
crisp = best_position[0]
variable_cost = np.arange(1, 4.5, 0.5)
var = ctrl.Antecedent(variable_cost, 'variable1_cost')
var['low'] = fuzz.trimf(var.universe, [1, 1 , 2])
var['medium'] = fuzz.trimf(var.universe, [1.5, 2, 3])
var['high'] = fuzz.trimf(var.universe, [3, 4, 4])
fuzzy_value_low_var = fuzz.interp_membership(var.universe, var['low'].mf, crisp)
fuzzy_value_medium_var = fuzz.interp_membership(var.universe, var['medium'].mf, crisp)
fuzzy_value_high_var = fuzz.interp_membership(var.universe, var['high'].mf, crisp)
print(f"Fuzzy value for 'low' at {crisp}: {fuzzy_value_low_var}")
print(f"Fuzzy value for 'medium' at {crisp}: {fuzzy_value_medium_var}")
print(f"Fuzzy value for 'high' at {crisp}: {fuzzy_value_high_var}")
```

```
Fuzzy value for 'low' at 4.0: 0.0
Fuzzy value for 'medium' at 4.0: 0.0
Fuzzy value for 'high' at 4.0: 1.0
```

we notice here that the value of the decision variable is high with 1 to the high membership function.

Defuzzification

After getting the membership functions, we convert them to a defuzzified variable by getting some random values and comparing their relation by the membership functions with a random number to filter them based on the fuzzy logic condition and get the mean of the random values to get more precise value for the parameter.

```
simulations = 20
defuzzified_prices = []

for i in range(simulations):
    while True:
        x = np.random.uniform(0.5, 4)
        u_high = fuzz.interp_membership(var.universe, var['high'].mf, x)
        alpha = np.random.uniform(0, 1)
        if alpha <= u_high:
            defuzzified_prices.append(x)
            break

average_defuzzified_price = np.mean(defuzzified_prices)
average_defuzzified_price = np.minimum(x_max[0] , average_defuzzified_price)
average_defuzzified_price = np.maximum(x_min[0], average_defuzzified_price)
print(f"Average 'defuzzified' price  {average_defuzzified_price:.2f}")
print("the maximized value of the profit would be after the deffuzified price is " + str(19606 * average_defuzzified_price + 67897 *best_position[1]  + 22632 * best_position[2] + 17749 * best_position[3])
```

```
Average 'defuzzified' price  3.71
the maximized value of the profit would be after the deffuzified price is 237523.02137079876
```

6-References:

1. https://rb.gy/aaz3oq .
2. https://www.tandfonline.com/doi/pdf/10.1080/08839514.2013.805596 .
3. https://mspace.lib.umanitoba.ca/server/api/core/bitstreams/a081528a-e903-4a78-85cf-9767758a2a8e/content .