## 1- Genetic Algorithm for Continuous Optimization

#### Algorithm

1. **Initialize Population**: Randomly generate bitstrings representing potential solutions.
2. **Decode**: Convert bitstrings to continuous values within given bounds.
3. **Evaluate**: Calculate the fitness of each solution.
4. **Selection**: Select parents using tournament selection.
5. **Crossover**: Perform crossover between two parents to produce offspring.
6. **Mutation**: Apply random mutation on offspring.
7. **Repeat**: Repeat steps 4-6 for several generations.
8. **Output**: Return the best solution and its fitness.

```
from numpy.random import randint, rand

def objective(x): return sum(i**2 for i in x)

def decode(bounds, n_bits, bitstring):
    largest = 2**n_bits - 1
    return [bounds[i][0] + (int(''.join(map(str,
bitstring[i*n_bits:(i+1)*n_bits])), 2) / largest) * (bounds[i][1] -
bounds[i][0]) for i in range(len(bounds))]

def selection(pop, scores, k=3):
    return min([pop[randint(len(pop))] for _ in range(k)], key=lambda p:
scores[pop.index(p)])

def crossover(p1, p2, r_cross):
```

```
    if rand() < r_cross:
        pt = randint(1, len(p1)-1)
        return [p1[:pt] + p2[pt:], p2[:pt] + p1[pt:]]
    return [p1, p2]

def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        if rand() < r_mut:
            bitstring[i] = 1 - bitstring[i]

def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop,
r_cross, r_mut):
    pop = [randint(0, 2, n_bits * len(bounds)).tolist() for _ in
range(n_pop)]
    best, best_eval = pop[0], objective(decode(bounds, n_bits, pop[0]))

    for gen in range(n_iter):
        scores = [objective(decode(bounds, n_bits, p)) for p in pop]
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(f">iter {gen}, new best f({decode(bounds, n_bits,
pop[i])}) = {scores[i]:.6f}")

        selected = [selection(pop, scores) for _ in range(n_pop)]
        pop = [child for i in range(0, n_pop, 2) for child in
crossover(selected[i], selected[i+1], r_cross)]
        for c in pop: mutation(c, r_mut)

    return best, best_eval

bounds = [[-5.0, 5.0], [-5.0, 5.0]]
```

```
n_iter, n_bits, n_pop, r_cross = 100, 16, 100, 0.9
r_mut = 1.0 / (n_bits * len(bounds))

best, score = genetic_algorithm(objective, bounds, n_bits, n_iter,
n_pop, r_cross, r_mut)
decoded_best = decode(bounds, n_bits, best)
print(f"\nGenetic algorithm completed\nBest solution:
{decoded_best}\nFitness score: {score:.5f}")
```

OUTPUT

```
>iter 0, new best f([-2.047271728515625, -1.97540283203125]) =
8.093538
>iter 0, new best f([0.15594482421875, -1.57745361328125]) =
2.512679
>iter 0, new best f([-0.55755615234375, -1.076812744140625]) =
1.470395
...
Genetic algorithm completed
Best solution: [-0.000152587890625, 0.0]
Fitness score: 0.00000
```

## 2 - Genetic Algorithm for Binary Optimization:

**Algorithm:**

1. **Initialize** a population of random binary strings.
2. **Evaluate** fitness for each individual.
3. **For each generation**:
    - **Select** parents using tournament selection.
    - **Crossover** parents to produce offspring.
    - **Mutate** offspring.
    - **Evaluate** new population's fitness.
4. **Replace** old population with new offspring.
5. **Return** the best solution and its fitness.

```
from numpy.random import randint, rand

def onemax(x): return -sum(x)

def selection(pop, scores, k=3):
    return min([pop[randint(len(pop))] for _ in range(k)], key=lambda p:
scores[pop.index(p)])

def crossover(p1, p2, r_cross):
    if rand() < r_cross:
        pt = randint(1, len(p1)-1)
        return [p1[:pt] + p2[pt:], p2[:pt] + p1[pt:]]
    return [p1, p2]

def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        if rand() < r_mut:
```

```
            bitstring[i] = 1 - bitstring[i]

def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    best, best_eval = pop[0], objective(pop[0])

    for gen in range(n_iter):
        scores = [objective(c) for c in pop]
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(f">iter {gen}, new best f({pop[i]}) = {scores[i]:.3f}")

        selected = [selection(pop, scores) for _ in range(n_pop)]
        pop = [child for i in range(0, n_pop, 2) for child in
crossover(selected[i], selected[i+1], r_cross)]
        for c in pop: mutation(c, r_mut)

    return best, best_eval

n_iter, n_bits, n_pop, r_cross = 100, 20, 100, 0.9
r_mut = 1.0 / float(n_bits)

print(f'Starting genetic algorithm\n')
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop,
r_cross, r_mut)
print(f'\nGenetic algorithm completed\nBest solution: {best}\nFitness
score: {score:.5f}')

OUTPUT
>iter 0, new best f([1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1])
= -14.000
```

>iter 0, new best f([1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0]) = -16.000
>iter 1, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0]) = -17.000
>iter 1, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1]) = -18.000
>iter 2, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]) = -19.000
>iter 4, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Genetic algorithm completed
Best solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Fitness score: -20.00000

## 3 - Simulated Annealing:

**Algorithm:**

1. **Initialize** a random starting point within bounds.
2. **Evaluate** the initial solution.
3. **Repeat** for a given number of iterations:
    - **Generate** a candidate solution by perturbing the current solution.
    - **Evaluate** the candidate solution.
    - If the candidate is better than the best solution, update the best solution.
    - **Calculate** the temperature decay.
    - **Metropolis criterion**: If the candidate solution is worse, accept it based on the temperature and a random probability.
4. **Return** the best solution and its evaluation.

```
from numpy import asarray, exp
from numpy.random import randn, rand, seed

def objective(x): return x[0]**2

def simulated_annealing(objective, bounds, n_iter, step_size, temp):
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:,
0])
    best_eval = objective(best)
    curr, curr_eval = best, best_eval
    for i in range(1, n_iter+1):
        candidate = curr + randn(len(bounds)) * step_size
        candidate_eval = objective(candidate)
        if candidate_eval < best_eval:
            best, best_eval = candidate, candidate_eval
```

```
            print(f'>iteration {i}: f({best}) = {best_eval:.5f}')
        diff, t = candidate_eval - curr_eval, temp / (i + 1)
        if diff < 0 or rand() < exp(-diff / t):
            curr, curr_eval = candidate, candidate_eval
    return best, best_eval

seed(1)
bounds = asarray([[-5.0, 5.0]])
best, score = simulated_annealing(objective, bounds, 1000, 0.1, 10)
print(f'\nBest solution: {best}\nFitness score: {score:.5f}')
```

OUTPUT

```
Starting simulated annealing algorithm

>iteration 34: f([-0.78753544]) = 0.62021
>iteration 35: f([-0.76914239]) = 0.59158
>iteration 37: f([-0.68574854]) = 0.47025
>iteration 39: f([-0.64797564]) = 0.41987
>iteration 40: f([-0.58914623]) = 0.34709
>iteration 41: f([-0.55446029]) = 0.30743
>iteration 42: f([-0.41775702]) = 0.17452
>iteration 43: f([-0.35038542]) = 0.12277
>iteration 50: f([-0.15799045]) = 0.02496

Simulated annealing completed

Best solution: [0.00013605]
Fitness score: 0.00000
```

## 4 - Ant Colony Optimization (ACO)

**Algorithm:**

1. **Initialize**: Set initial pheromone levels and distances.
2. **Ant Movement**: Each ant picks a city based on pheromone and visibility. Repeat until all cities are visited.
3. **Compute Path Length**: Calculate the total distance of the ant's tour.
4. **Pheromone Update**: Update pheromone levels based on the tour quality.
5. **Iterate**: Repeat for a set number of iterations, finding the best tour.

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

def plot_graph(g, title="", highlight_edges=[]):
    pos = nx.get_node_attributes(g, "pos")
    plt.figure(figsize=(10, 10))
    nx.draw(g, pos=pos, with_labels=True, width=2)
    weights = nx.get_edge_attributes(g, "weight")
    nx.draw_networkx_edge_labels(g, pos, edge_labels=weights,
label_pos=0.4)
    nx.draw_networkx_edges(g, pos, edgelist=highlight_edges,
edge_color="r", width=3)
    plt.title(title)
    plt.show()

def zero_divide(a, b):
```

```
    return np.divide(a, b, out=np.zeros_like(a), where=b != 0)

class ACO_TSP:
    def __init__(self, g, n_ants=100, alpha=1, beta=5, Q=100, rho=0.6):
        self.g = g
        self.n_nodes = len(g.nodes)
        self.visibility = zero_divide(np.ones_like(nx.to_numpy_array(g)),
nx.to_numpy_array(g))
        self.n_ants = n_ants
        self.alpha = alpha
        self.beta = beta
        self.Q = Q
        self.rho = rho
        self.phe_trail = np.ones((self.n_nodes, self.n_nodes))

    def compute_prob(self, visited):
        prob = self.phe_trail**self.alpha * self.visibility**self.beta
        prob[:, list(visited)] = 0
        prob_sum = prob.sum(-1, keepdims=True)
        return zero_divide(prob, prob_sum)

    def initialize(self):
        self.ant_pos = np.random.choice(list(self.g.nodes), self.n_ants)

    def path_length(self, path):
        edge_weights = nx.get_edge_attributes(self.g, "weight")
        return sum(edge_weights[tuple(sorted(edge))] for edge in path)

    def ant_tour(self, k):
        current = self.ant_pos[k]
        visited = {current}
        path = []
```

```
        while len(visited) < self.n_nodes:
            prob = self.compute_prob(visited)
            current = np.random.choice(self.n_nodes, p=prob[current])
            visited.add(current)
            path.append((current, current))
        return path

    def update_pheromone(self):
        d_phe_trail = np.zeros_like(self.phe_trail)
        for k in range(self.n_ants):
            if len(self.paths[k]) == self.n_nodes - 1:
                for i, j in self.paths[k]:
                    d_phe_trail[i, j] = d_phe_trail[j, i] = self.Q /
self.path_lengths[k]
        self.phe_trail = self.rho * self.phe_trail + d_phe_trail

    def run(self, n_iter=50):
        self.initialize()
        for _ in range(n_iter):
            self.paths = [self.ant_tour(k) for k in range(self.n_ants)]
            self.path_lengths = [self.path_length(path) for path in
self.paths]
            self.update_pheromone()
            print(f"Shortest Path Length: {min(self.path_lengths)}")

    @property
    def min_path_length(self):
        return min(self.path_lengths)

    @property
    def min_path(self):
        return self.paths[np.argmin(self.path_lengths)]
```

```
# Generate random weighted graph (Example)
def generate_random_weighted_graph(n, min_weight=1,
max_weight=10):
    G = nx.complete_graph(n)
    for u, v in G.edges:
        G[u][v]['weight'] = np.random.randint(min_weight, max_weight)
    return G

# Main Execution
np.random.seed(3)
g = generate_random_weighted_graph(10)
plot_graph(g, "Graph for TSP")

aco_tsp = ACO_TSP(g, n_ants=10, alpha=3, beta=5, Q=10, rho=0.1)
aco_tsp.run(n_iter=50)

plot_graph(g, "Shortest Path Found by ACO for TSP",
aco_tsp.min_path)
```

**OUTPUT - DIAGRAM**

## 5 - Particle Swarm Optimization (PSO)

Algorithm (Shortened):

1. Initialize the swarm: Randomly initialize `n` particles with positions and velocities within the given bounds.
2. Set hyperparameters: Choose `dim` (dimensions), `minx` (lower bound), `maxx` (upper bound), and optimization constants (`w`, `c1`, `c2`).
3. Main loop: Repeat for `max_iter` iterations:
    - For each particle, update its velocity based on its best position and the global best.
    - Update the particle's position and fitness.
    - If the new position is better than its previous best, update its best position.
    - If the particle's best fitness is better than the global best, update the global best.
4. Return the global best position after `max_iter` iterations.

```
import random
import sys

def fitness_sphere(position):
    return sum(x**2 for x in position)

class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [self.rnd.uniform(minx, maxx) for _ in range(dim)]
        self.velocity = [self.rnd.uniform(minx, maxx) for _ in range(dim)]
        self.best_pos = self.position[:]
```

```
        self.best_fitness = fitness(self.position)

    def update_velocity(self, global_best_pos, w, c1, c2):
        r1, r2 = self.rnd.random(), self.rnd.random()
        self.velocity = [
            w * v + c1 * r1 * (bp - p) + c2 * r2 * (gb - p)
            for v, bp, p, gb in zip(self.velocity, self.best_pos, self.position,
global_best_pos)
        ]

    def update_position(self, minx, maxx):
        self.position = [max(minx, min(p + v, maxx)) for p, v in
zip(self.position, self.velocity)]
        self.fitness = fitness_sphere(self.position)

    def update_best(self):
        if self.fitness < self.best_fitness:
            self.best_pos, self.best_fitness = self.position[:], self.fitness

def pso(fitness, max_iter, n, dim, minx, maxx):
    w, c1, c2 = 0.729, 1.49445, 1.49445
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]
    global_best_pos = min(swarm, key=lambda p:
p.best_fitness).best_pos
    global_best_fitness = min(swarm, key=lambda p:
p.best_fitness).best_fitness

    for Iter in range(max_iter):
        if Iter % 10 == 0 and Iter > 1:
            print(f"Iter = {Iter} best fitness = {global_best_fitness:.3f} Best
position: {[f'{x:.6f}' for x in global_best_pos]}")
```

```
        for particle in swarm:
            particle.update_velocity(global_best_pos, w, c1, c2)
            particle.update_position(minx, maxx)
            particle.update_best()
            if particle.best_fitness < global_best_fitness:
                global_best_pos, global_best_fitness = particle.best_pos,
particle.best_fitness

    return global_best_pos

# Driver code
dim, max_iter, num_particles = 3, 100, 50
print("\nStarting PSO algorithm\n")
best_position = pso(fitness_sphere, max_iter, num_particles, dim,
-10.0, 10.0)
print("\nPSO completed\n")
print("\nBest solution found:")
print([f"{x:.6f}" for x in best_position])
fitnessVal = fitness_sphere(best_position)
print(f"fitness of best solution = {fitnessVal:.6f}")
```

OUTPUT

```
Starting PSO algorithm

Iter = 10 best fitness = 9.700 Best position: ['-2.963075', '-0.926583',
'0.248887']
Iter = 20 best fitness = 9.700 Best position: ['-2.963075', '-0.926583',
'0.248887']
Iter = 30 best fitness = 1.100 Best position: ['-0.411780', '-0.854872',
'0.446743']
```

Iter = 40 best fitness = 0.873 Best position: ['-0.417564', '-0.829184', '0.105651']
Iter = 50 best fitness = 0.855 Best position: ['-0.405815', '-0.829811', '0.036998']
Iter = 60 best fitness = 0.851 Best position: ['-0.403051', '-0.829691', '0.013340']
Iter = 70 best fitness = 0.850 Best position: ['-0.402990', '-0.829325', '0.015951']
Iter = 80 best fitness = 0.850 Best position: ['-0.402987', '-0.829222', '0.020165']
Iter = 90 best fitness = 0.850 Best position: ['-0.402987', '-0.829218', '0.020344']

PSO completed

Best solution found:
['-0.402987', '-0.829218', '0.020351']
fitness of best solution = 0.850415

## 6 - Gray Wolf Optimization (GWO)

**Initialize Population**:

- Randomly initialize $n$ grey wolves $X_i$ within bounds $[minx, maxx]$.

**Evaluate Fitness**:

- Calculate fitness for each wolf.
- Identify $X_\alpha$, $X_\beta$, and $X_\gamma$ (the top 3 best wolves).

**Iterate** for $T$ iterations:

- Update the coefficient $a = 2(1 - \frac{t}{T})$.
- For each wolf:
  - Update position using:

$$X_i = \frac{X_\alpha + X_\beta + X_\gamma}{3} - A \times |C \times X_i - X_{best}|$$

  - Update fitness and select new best positions if applicable.

**Return**: Best solution ($X_\alpha$) after $T$ iterations.

```
import random

# Sphere fitness function
def fitness_sphere(position):
    return sum(x**2 for x in position)

# Grey Wolf Optimization (GWO) class
def gwo(fitness, max_iter, n, dim, minx, maxx):
    rnd = random.Random(0)
    population = [wolf(fitness, dim, minx, maxx, i) for i in range(n)]

    # Sort population based on fitness
    population.sort(key=lambda w: w.fitness)
```

```
    alpha, beta, gamma = population[:3]

    for Iter in range(max_iter):
        if Iter % 10 == 0 and Iter > 1:
            print(f"Iter = {Iter}, best fitness = {alpha.fitness:.3f}, Best
position = {[f'{x:.6f}' for x in alpha.position]}")

        a = 2 * (1 - Iter / max_iter)  # Linearly decrease a

        # Update each wolf's position
        for wolf in population:
            A, C = [a * (2 * rnd.random() - 1) for _ in range(3)], [2 *
rnd.random() for _ in range(3)]
            Xnew = [(alpha.position[i] - A[0] * abs(C[0] * alpha.position[i] -
wolf.position[i]) +
                     beta.position[i] - A[1] * abs(C[1] * beta.position[i] -
wolf.position[i]) +
                     gamma.position[i] - A[2] * abs(C[2] * gamma.position[i] -
wolf.position[i])) / 3
                    for i in range(dim)]

            # Fitness of new position
            fnew = fitness(Xnew)
            if fnew < wolf.fitness:
                wolf.position, wolf.fitness = Xnew, fnew

        # Update alpha, beta, gamma
        population.sort(key=lambda w: w.fitness)
        alpha, beta, gamma = population[:3]

    return alpha.position
```

```
# Wolf class
class wolf:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.position = [random.uniform(minx, maxx) for _ in range(dim)]
        self.fitness = fitness(self.position)

# Driver code
dim = 3
fitness = fitness_sphere
num_particles = 10
max_iter = 50

print('Starting graywolf algorithm\n')
best_position = gwo(fitness, max_iter, num_particles, dim, -10.0, 10.0)
print('\nGraywolf algorithm completed\n')

print("\nBest solution found:")
print([f"{x:.6f}" for x in best_position])
print(f"Fitness of best solution = {fitness(best_position):.6f}")
```

OUTPUT

```
Starting graywolf algorithm

Iter = 10, best fitness = 0.012, Best position = ['-0.044360', '0.084065',
'-0.050042']
Iter = 20, best fitness = 0.000, Best position = ['-0.004129', '0.005473',
'0.000494']
Iter = 30, best fitness = 0.000, Best position = ['-0.001647', '0.001247',
'-0.000390']
Iter = 40, best fitness = 0.000, Best position = ['-0.000896', '0.001025',
'-0.000212']
```

Graywolf algorithm completed

Best solution found:
['-0.000871', '0.000903', '-0.000206']
Fitness of best solution = 0.000002

## 7 - Tabu Search

**Shortest Tabu Search Algorithm:**

1. **Initialize**:
    - Generate an initial solution randomly.
    - Calculate its objective value.
2. **Create Tabu List**:
    - Generate all possible swap pairs (neighbors).
3. **Main Loop**:
    - For each iteration:
        - Evaluate neighbors (swap jobs).
        - Select best non-tabu or aspiration move.
        - Update best solution if improvement is found.
4. **Update Tabu List**:
    - Add the selected move to the tabu list with an expiration time.
5. **Repeat** until termination (e.g., max iterations or no improvement).
6. **Output Best Solution**.

```
import pandas as pd
import random as rd
from itertools import combinations
```

```
class TS:
    def __init__(self, path, seed, tabu_tenure):
        self.instance_dict = pd.read_excel(path, names=['Job', 'weight',
"processing_time", "due_date"], index_col=0).to_dict('index')
        self.seed = seed
        self.tabu_tenure = tabu_tenure
        self.initial_solution = self._get_initial_solution()
        self.best_solution, self.best_obj_value = self._tabu_search()

    def _get_initial_solution(self):
        solution = list(range(1, len(self.instance_dict) + 1))
        rd.seed(self.seed)
        rd.shuffle(solution)
        return solution

    def _obj_fun(self, solution):
        t = obj_value = 0
        for job in solution:
            C_i = t + self.instance_dict[job]["processing_time"]
            T_i = max(0, C_i - self.instance_dict[job]["due_date"])
            obj_value += self.instance_dict[job]["weight"] * T_i
            t = C_i
        return obj_value

    def _swap_move(self, solution, i, j):
        solution = solution.copy()
        solution[solution.index(i)], solution[solution.index(j)] =
solution[solution.index(j)], solution[solution.index(i)]
        return solution

    def _get_tabu_structure(self):
```

```
        return {swap: {'tabu_time': 0, 'move_value': float('inf')} for swap in
combinations(self.instance_dict.keys(), 2)}

    def _tabu_search(self):
        tabu_structure = self._get_tabu_structure()
        best_solution = self.initial_solution
        best_obj_value = self._obj_fun(best_solution)
        current_solution = best_solution
        current_obj_value = best_obj_value
        iter_count, terminate = 1, 0

        while terminate < 100:
            for move in tabu_structure:
                candidate_solution = self._swap_move(current_solution,
move[0], move[1])
                tabu_structure[move]['move_value'] =
self._obj_fun(candidate_solution)

            while True:
                best_move = min(tabu_structure, key=lambda x:
tabu_structure[x]['move_value'])
                move_value, tabu_time =
tabu_structure[best_move]["move_value"],
tabu_structure[best_move]["tabu_time"]

                if tabu_time < iter_count:
                    current_solution = self._swap_move(current_solution,
best_move[0], best_move[1])
                    current_obj_value = self._obj_fun(current_solution)

                    if move_value < best_obj_value:
```

```
                    best_solution, best_obj_value = current_solution,
current_obj_value
                    terminate = 0
                else:
                    terminate += 1

                tabu_structure[best_move]['tabu_time'] = iter_count +
self.tabu_tenure
                iter_count += 1
                break

            elif move_value < best_obj_value:
                current_solution = self._swap_move(current_solution,
best_move[0], best_move[1])
                best_solution = current_solution
                best_obj_value = self._obj_fun(current_solution)
                terminate = 0
                iter_count += 1
                break

            tabu_structure[best_move]["move_value"] = float('inf')

        return best_solution, best_obj_value

print("Starting Tabu search\n")
test = TS(path="Instance_10.xlsx", seed=2012, tabu_tenure=3)
```

**OUTPUT**

```
Starting Tabu search

Iteration 1: Best_objvalue: 29.220000000000002
```

Iteration 2: Best_objvalue: 21.62
...
Tabu search completed

Performed iterations: 110
Best found Solution: [3, 2, 1, 4, 8, 10, 5, 9, 7, 6] , Objvalue:
13.240000000000002