simplest_code

February 4, 2024

[2]:
```
import random
import math
import matplotlib.pyplot as plt
import numpy as np
```

1 Monte Carlo A

[3]:
```
def f(x):
    return x**2

def monte_carlo_integration(num_samples, a, b):
    integral_sum = 0

    for _ in range(num_samples):
        x = random.uniform(a, b)
        integral_sum += f(x)

    estimated_integral = ((b - a) / num_samples) * integral_sum
    return estimated_integral

num_samples = 10000  # You can adjust the number of samples for more accuracy
a, b = 0, 1  # Integration limits
estimated_integral = monte_carlo_integration(num_samples, a, b)

print(f"Estimated Integral: {estimated_integral}")
```

```
Estimated Integral: 0.3315121609624472
```

2 Monte Carlo Importance Sampling

[6]:
```
def f(x):
    return 1 / (x**2 + 1)

def sampling_distribution(x):
    return 2 / (2 + x)

def original_distribution():
```

```
    x = random.uniform(a, b)
    return x

def importance_sampling(num_samples, a, b):
    integral_sum = 0

    for _ in range(num_samples):
        x = original_distribution()
        weight = sampling_distribution(x) / (b - a)
        integral_sum += f(x) / weight

    estimated_integral = integral_sum / num_samples
    return estimated_integral

num_samples = 10000  # You can adjust the number of samples for more accuracy
a, b = -1, 1  # Integration limits
estimated_integral = importance_sampling(num_samples, a, b)

print(f"Estimated Integral: {estimated_integral}")
```

```
Estimated Integral: 1.5633994889690292
```

3 Rejection Sampling

[8]:
```
# Target distribution (normal distribution)
def target_distribution(x):
    return (1 / (math.sqrt(2 * math.pi))) * math.exp(-0.5 * x**2)

# Proposal distribution (uniform distribution)
def proposal_distribution(a, b):
    return random.uniform(a, b)

def rejection_sampling(a, b):
    while True:
        x = proposal_distribution(a, b)
        accept_prob = target_distribution(x) / (1 / (b - a))

        if random.uniform(0, 1) < accept_prob:
            sample = x
            break

    return sample

a, b = -3, 3  # Range for the uniform proposal distribution
sample = rejection_sampling(a, b)
```

4 Inverse Sampling

[ ]:
```
# Generate a random number from an exponential distribution with rate lambda
def inverse_sampling_exponential():
    # Generate a random number sampled from U(0, 1)
    u = random.random()

    # Use the inverse of the CDF for the exponential distribution to transform␣
 ↪the random number
    # Inverse CDF for exponential distribution: x = -ln(1 - u) / lambda
    lambda_val = 0.5
    sampled_value = -math.log(1 - u) / lambda_val

    return sampled_value

# Generate numbers following an exponential distribution
random_number = inverse_sampling_exponential()
print(f"Random Number: {random_number:.4f}")
```

5 Metropolis-Hastings

[ ]:
```
# Define the target distribution (a simple discrete distribution)
def target_distribution(x):
    if x == 1:
        return 0.3
    elif x == 2:
        return 0.4
    else:
        return 0.3

# Proposal distribution (simple random walk)
def proposal_distribution(x):
    return x + random.choice([-1, 1])

def metropolis_hastings(num_samples, initial_sample):
    samples = [initial_sample]

    for _ in range(num_samples - 1):
        current_sample = samples[-1]
        proposed_sample = proposal_distribution(current_sample)

        # Calculate acceptance ratio
        acceptance_ratio = min(1, target_distribution(proposed_sample) /␣
 ↪target_distribution(current_sample))

        # Accept or reject the proposal
```

```
        if random.uniform(0, 1) < acceptance_ratio:
            samples.append(proposed_sample)
        else:
            samples.append(current_sample)

    return samples

num_samples = 100
initial_sample = 1  # Start from state 1
samples = metropolis_hastings(num_samples, initial_sample)
```

6 SSA

[ ]:
```
# Initialize the system
population = 10
time = 0
time_points = [time]
population_counts = [population]

# Simulation parameters
end_time = 100.0

while time < end_time:
    # Define reaction constants
    birth_rate = 2.0
    death_rate = 0.02

    # Calculate the propensity functions
    a_birth = birth_rate
    a_death = death_rate * population
    rxn_as = np.array([a_birth, a_death])
    a_total = np.sum(rxn_as)

    if a_total != 0: # can't divide by 0

        # Calculate time step (tau) from exponential distribution
        tau = -np.log(np.random.uniform(0,1)) / a_total

        # Determine the next reaction to occur
        possible_rxns = np.arange(len(rxn_as))
        i = np.random.choice(possible_rxns, p=rxn_as/a_total)

        # Apply rxn
        if i == 0:
            population += 1
        elif i == 1:
```

```
            population -= 1

        # Update time and counts
        time += tau
        population_counts.append(population)
        time_points.append(time)
    else: # a0 == 0, no more rxns
        time = end_time
        population_counts.append(population)
        time_points.append(time)

# Plot the results
plt.plot(time_points, population_counts)
```

7 Tau leaping

[ ]:
```
# Initialize the system
population = 10
time = 0
time_points = [time]
population_counts = [population]

# Simulation parameters
end_time = 10.0
tau = 0.1

while time < end_time:
    # Define reaction constants
    birth_rate = 2.0
    death_rate = 0.02

    # Calculate the propensity functions
    a_birth = birth_rate
    a_death = death_rate * population

    # Sample the number of reactions for each channel from Poisson distribution
    num_births = 0
    num_deaths = 0
    if (a_birth*tau > 0):
        num_births = np.random.poisson(a_birth * tau)
    if (a_death*tau > 0):
        num_deaths = np.random.poisson(a_death * tau)

    # Update time and population count
    time += tau
    population += num_births - num_deaths
```

```
    population_counts.append(population)
    time_points.append(time)

# Plot the results
plt.step(time_points, population_counts)
```

8 PSO

[576]:
```
def f(x):
    return x**2

# Define the PSO parameters
num_particles = 10
max_iterations = 10
c1 = 1.5  # Cognitive parameter
c2 = 1.5  # Social parameter
w = 0.7   # Inertia weight

# Initialize the particles
particles = []
for _ in range(num_particles):
    position = random.uniform(-10, 10)  # Random initial position
    velocity = random.uniform(-1, 1)    # Random initial velocity
    personal_best_position = position
    personal_best_value = f(position)
    particles.append([position, velocity, personal_best_position,␣
 ↪personal_best_value])

# Initialize the global best-known position and value
global_best_position = particles[0][0]
global_best_value = particles[0][3]

# PSO optimization loop
for iteration in range(max_iterations):
    for i in range(num_particles):
        particle = particles[i]
        position, velocity, personal_best_position, personal_best_value =␣
 ↪particle

        # 1. Update velocity and position
        r1, r2 = random.random(), random.random()
        velocity = w * velocity + c1 * r1 * (personal_best_position - position)␣
 ↪+ c2 * r2 * (global_best_position - position)
        position = position + velocity

        # 2. Evaluate the objective function at the new position
```

```
        value = f(position)

        # 3. Update personal best if needed
        if value < personal_best_value:
            personal_best_position = position
            personal_best_value = value

        # 4. Update global best if needed
        if value < global_best_value:
            global_best_position = position
            global_best_value = value

        # 5. Update the particle's information
        particles[i] = [position, velocity, personal_best_position,␣
↪personal_best_value]

    print(f"Iteration {iteration + 1}: Global Best Value = {global_best_value:.
↪4f}")

print(f"Optimal Solution: x = {global_best_position}, f(x) = {global_best_value:
↪.4f}")
```

```
Iteration 1: Global Best Value = 0.1242
Iteration 2: Global Best Value = 0.1242
Iteration 3: Global Best Value = 0.0003
Iteration 4: Global Best Value = 0.0003
Iteration 5: Global Best Value = 0.0003
Iteration 6: Global Best Value = 0.0003
Iteration 7: Global Best Value = 0.0003
Iteration 8: Global Best Value = 0.0003
Iteration 9: Global Best Value = 0.0003
Iteration 10: Global Best Value = 0.0003
Optimal Solution: x = -0.01583582034806863, f(x) = 0.0003
```

9 ACO

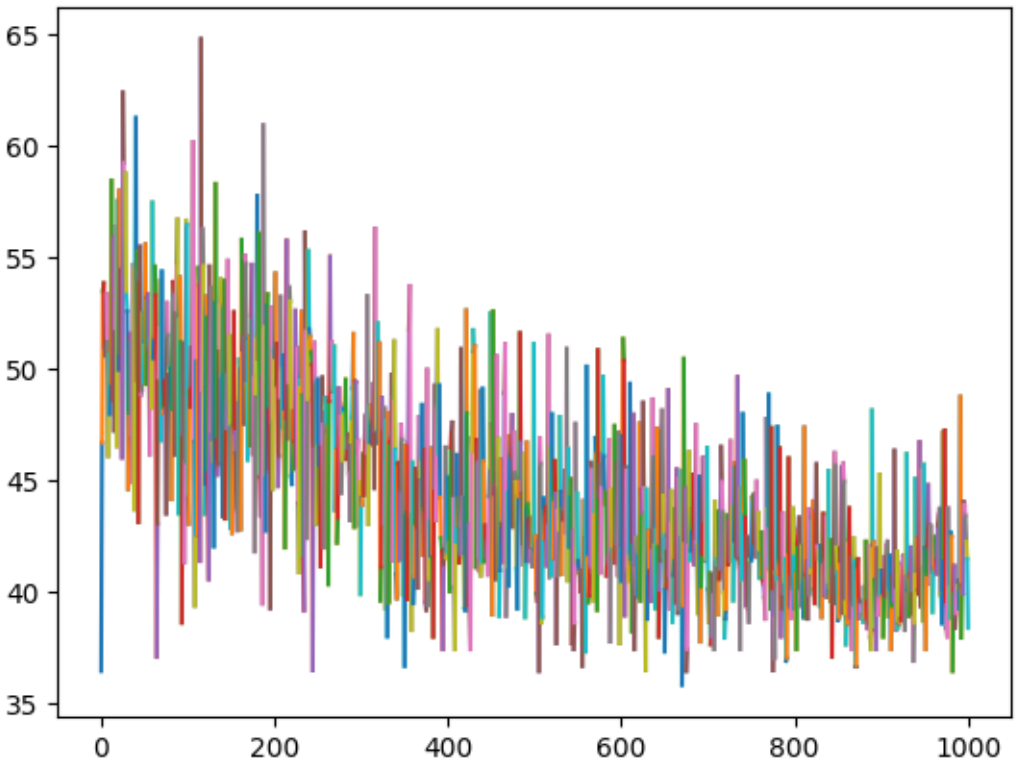
[548]:
```
# Define the distance matrix between cities (1st city to 2nd city)
distance_matrix = [
    [0, 3.605551275463989, 5.0990195135927845, 7.211102550927978, 10.
↪816653826391969, 8.54400374531753, 12.806248474865697, 13.038404810405298,␣
↪13.341664064126334, 10.0],
    [3.605551275463989, 0, 3.605551275463989, 4.123105625617661, 7.
↪615773105863909, 5.0990195135927845, 9.219544457292887, 9.848857801796104,␣
↪11.0, 8.54400374531753],
    [5.0990195135927845, 3.605551275463989, 0, 3.1622776601683795, 6.
↪4031242374328485, 7.280109889280518, 9.486832980505138, 8.48528137423857, 8.
↪246211251235321, 5.0990195135927845],
```

```
    [7.211102550927978, 4.123105625617661, 3.1622776601683795, 0, 3.
 ↪605551275463989, 5.0, 6.324555320336759, 5.830951894845301, 7.
 ↪0710678118654755, 5.656854249492381],
    [10.816653826391969, 7.615773105863909, 6.4031242374328485, 3.
 ↪605551275463989, 0, 6.324555320336759, 4.123105625617661, 2.23606797749979,␣
 ↪5.0, 6.082762530298219],
    [8.54400374531753, 5.0990195135927845, 7.280109889280518, 5.0, 6.
 ↪324555320336759, 0, 5.385164807134504, 8.06225774829855, 11.180339887498949,␣
 ↪10.63014581273465],
    [12.806248474865697, 9.219544457292887, 9.486832980505138, 6.
 ↪324555320336759, 4.123105625617661, 5.385164807134504, 0, 4.242640687119285,␣
 ↪8.602325267042627, 10.198039027185569],
    [13.038404810405298, 9.848857801796104, 8.48528137423857, 5.
 ↪830951894845301, 2.23606797749979, 8.06225774829855, 4.242640687119285, 0, 4.
 ↪47213595499958, 7.0710678118654755],
    [13.341664064126334, 11.0, 8.246211251235321, 7.0710678118654755, 5.0, 11.
 ↪180339887498949, 8.602325267042627, 4.47213595499958, 0, 4.242640687119285],
    [10.0, 8.54400374531753, 5.0990195135927845, 5.656854249492381, 6.
 ↪082762530298219, 10.63014581273465, 10.198039027185569, 7.0710678118654755,␣
 ↪4.242640687119285, 0]
]

# ACO parameters
num_ants = 3
num_iterations = 1000
initial_pheromone = 0.001
evaporation_rate = 0.01
pheromone_deposit_rate = 1.0
alpha = 3.0 # Pheromone Importance
beta = 1.0 # Heuristic Importance
exploitation_prob_q = 0.05 # Exploitation vs Exploration probability

# Initialize pheromone levels
num_cities = len(distance_matrix)
pheromone_matrix = [[initial_pheromone+1] * num_cities for _ in␣
 ↪range(num_cities)]

# ACO main loop
for iteration in range(num_iterations):

    ant_tours = []
    # 1. Loop through ants
    for ant in range(num_ants):
        # 2. randomly initialize ant at random city
        start_city = random.randint(0, num_cities - 1)
        tour = [start_city]
```

```
        # 3. Loop through remaining cities
        while len(tour) < num_cities:
            remaining_cities = [i for i in range(num_cities) if i not in tour]
            probabilities = []

            # 4. Decide Exploitation vs Exploration
            if random.uniform(0, 1) > exploitation_prob_q:
                # Explore based on pheromones
                for city in remaining_cities:
                    pheromone = pheromone_matrix[start_city][city]
                    distance = distance_matrix[start_city][city]
                    attractiveness = (pheromone ** alpha) * ((1 / distance) **␣
↪beta)
                    probabilities.append((city, attractiveness))
                total_attractiveness = sum(attractiveness for _, attractiveness␣
↪in probabilities)
                probabilities = [(city, attractiveness / total_attractiveness)␣
↪for city, attractiveness in probabilities]

                next_city = np.random.choice([city for city, _ in␣
↪probabilities], p=[prob for _, prob in probabilities])
            else:
                # Exploit based on pure heuristic
                min_distance = max(max(distance_matrix)) # initialize␣
↪min_distance
                for city in remaining_cities:
                    if min_distance >= distance_matrix[start_city][city]:
                        min_distance = distance_matrix[start_city][city]
                        next_city = city

            # 5. Local update pheromone levels to incentivize new trails
            pheromone_matrix[start_city][next_city] = (1 -␣
↪evaporation_rate)*pheromone_matrix[start_city][next_city] +␣
↪(evaporation_rate * initial_pheromone)
            pheromone_matrix[next_city][start_city] = (1 -␣
↪evaporation_rate)*pheromone_matrix[start_city][next_city] +␣
↪(evaporation_rate * initial_pheromone)
            tour.append(next_city)
            start_city = next_city

        ant_tours.append(tour)

    # for plotting
    prev_best_distance = best_distance
    # 6. Find the best tour in this iteration
```

```
    best_tour = min(ant_tours, key=lambda tour: sum(distance_matrix[tour[i -␣
↪1]][tour[i]] for i in range(1, num_cities)))
    best_distance = sum(distance_matrix[best_tour[i - 1]][best_tour[i]] for i␣
↪in range(1, num_cities))
    # 7. Trail update pheromone levels
    for i in range(1, num_cities):
        pheromone_matrix[best_tour[i-1]][best_tour[i]] +=␣
↪pheromone_deposit_rate / best_distance
        pheromone_matrix[best_tour[i]][best_tour[i-1]] +=␣
↪pheromone_deposit_rate / best_distance


    plt.plot([iteration, iteration + 1], [prev_best_distance, best_distance])

print("Best Tour Found:", best_tour)
print("Total Distance:", best_distance)
plt.show()
```

```
Best Tour Found: [3, 1, 0, 2, 9, 8, 7, 4, 6, 5]
Total Distance: 38.385810980638034
```



10 DE

[3]:
```
def f(x):
    return sum(x**2)

# DE Parameters
population_size=5  # Size of the population
min_bound=-10   # Lower bound for each parameter
max_bound=10   # Upper bound for each parameter
num_generations = 10
mutation=0.8 # Mutation Factor
crossover_probability=0.7  # Crossover_probability
dimensions = 2 # Number of dimensions of an individual

# 1. Initialize population with random solutions and their fitnesses
population = min_bound + np.random.rand(population_size,␣
 ↪dimensions)*(max_bound-min_bound)
fitnesses = np.array([f(individual) for individual in population])

# 2. Loop through generations
for generation in range(num_generations):
    # 3. Loop through population
    for individual in range(population_size):

        # 4. Choose 3 different individuals out of the population
        # Select indices of all individuals except the current individual
        idxs = [idx for idx in range(population_size) if idx != individual]
        a, b, c = population[np.random.choice(idxs, 3, replace = False)]

        # 5. Generate a mutant donor using the mutation factor and ensure still␣
 ↪in bounds
        mutant_donor = np.clip(a + mutation * (b - c), min_bound, max_bound)

        # 6. Crossover Operation
        # randomly select dimension points for crossover based on crossover␣
 ↪probability
        points_to_crossover = np.random.rand(dimensions) < crossover_probability
        # Ensure at least one dimension/parameter is modified
        if not np.any(points_to_crossover):
            points_to_crossover[np.random.randint(0, dimensions)] = True
        # Do Crossover
        trial_individual = np.where(points_to_crossover, mutant_donor,␣
 ↪population[individual])
        trial_individual_fitness = f(trial_individual)

        # 7. Individual is replaced is trial_individual has better fitness
        if trial_individual_fitness < fitnesses[individual]:
```

```
            fitnesses[individual] = trial_individual_fitness
            population[individual] = trial_individual

    # Find the best solution in the final population
    best_solution = min(population, key=f)
    best_fitness = f(best_solution)

    print(f"Best Solution: {best_solution}")
    print(f"Best Fitness: {best_fitness}")
```

```
Best Solution: [ 4.41175621 -0.52758748]
Best Fitness: 19.74194138421374
Best Solution: [ 4.41175621 -0.52758748]
Best Fitness: 19.74194138421374
Best Solution: [-0.49708186 -0.0597067 ]
Best Fitness: 0.25065526941514615
Best Solution: [-0.49708186 -0.0597067 ]
Best Fitness: 0.25065526941514615
Best Solution: [-0.49708186 -0.0597067 ]
Best Fitness: 0.25065526941514615
Best Solution: [0.11870417 0.48221743]
Best Fitness: 0.24662432951451072
Best Solution: [-0.14440907  0.01058759]
Best Fitness: 0.020966077069033456
Best Solution: [-0.00122927  0.10352288]
Best Fitness: 0.010718498212355987
Best Solution: [-0.00122927  0.10352288]
Best Fitness: 0.010718498212355987
Best Solution: [-0.05115475 -0.08012017]
Best Fitness: 0.009036049255503473
```

11 PBIL

[654]:
```
# Define the MaxOnes fitness function
def max_ones_fitness(binary_vector):
    return sum(binary_vector)

# PBIL parameters
vector_length = 10  # Length of the binary vector
population_size = 5
alpha = 0.2  # Alpha rate for updating probabilities (basically learning rate)
mutation_prob = 0.02  # Probability of mutating a bit
beta = 1.0 # Mutation strength
num_generations = 10

# Initialize the probability vector with equal probabilities
probability_vector = np.array([0.5] * vector_length)
```

```
# Main PBIL loop
for generation in range(num_generations):
    # 1. Generate a population based on the probability vector
    population = [[int(random.random() < p) for p in probability_vector] for _␣
 ↪in range(population_size)]

    # 2. Evaluate the fitness of each individual
    fitness_values = [max_ones_fitness(individual) for individual in population]

    # 3. Find the best individual in the population
    best_individual = np.array(population[fitness_values.
 ↪index(max(fitness_values))])

    # 4. Update the probability vector with alpha
    probability_vector = (1-alpha)*probability_vector  + (alpha*best_individual)

    # Mutate each element in probability vector based on mutation prob
    for i in range(vector_length):
        if random.random() < mutation_prob:
            probability_vector[i] = (1-beta)*probability_vector[i] +␣
 ↪(beta*random.uniform(0,1))

    # Find the best individual in the final population
    best_individual = max(population, key=max_ones_fitness)
    best_fitness = max_ones_fitness(best_individual)

    print("Best Individual:", best_individual, "Fitness:", best_fitness)
```

```
Best Individual: [1, 1, 1, 0, 1, 0, 1, 0, 1, 0] Fitness: 6
Best Individual: [1, 1, 0, 0, 1, 1, 1, 1, 1, 0] Fitness: 7
Best Individual: [1, 1, 1, 0, 1, 0, 1, 1, 1, 1] Fitness: 8
Best Individual: [1, 0, 1, 0, 1, 1, 1, 1, 1, 1] Fitness: 8
Best Individual: [1, 1, 1, 0, 1, 1, 1, 0, 1, 1] Fitness: 8
Best Individual: [1, 1, 1, 1, 1, 0, 1, 1, 1, 1] Fitness: 9
Best Individual: [0, 1, 1, 1, 1, 0, 1, 1, 0, 1] Fitness: 7
Best Individual: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] Fitness: 10
Best Individual: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] Fitness: 10
Best Individual: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1] Fitness: 10
```

12 PBILc

[619]:
```
def f(x):
    return -sum(x**2) # Negative since PBIL and PBILc finds the max

# PBIL parameters
vector_length = 2  # Length of the parameter vector
```

```
population_size = 10
alpha = 0.1  # Alpha rate for updating the parameter vector (learning rate)
num_generations = 10

# Initialize the parameter vector with random values
parameter_vector = np.array([0.5] * vector_length)
# Standard deviation for normal distribution sampling
standard_deviations = np.array([0.5] * vector_length)

# Main PBIL loop
for generation in range(num_generations):
    # 1. Generate a population based on the parameter vector
    population = [parameter_vector + np.random.normal(0, standard_deviations,␣
 ↪vector_length) for _ in range(population_size)]

    # 2. Evaluate the fitness of each individual
    fitness_values = [f(individual) for individual in population]

    # 3. Find the indices of the two best individuals and the worst individual
    best_indices = np.argsort(fitness_values)[-2:]  # Two best individuals
    worst_index = np.argmin(fitness_values)  # Worst individual

    # 4. Update the parameter vector using the two best and subtracting the␣
 ↪worst
    parameter_vector +=␣
 ↪alpha*(population[best_indices[0]]+population[best_indices[1]]-population[worst_index])

    # 5. Update standard deviation


    # Find the best individual in the final population
    best_fitness = f(parameter_vector)

    print("Best Parameter Vector:", parameter_vector, "Best Fitness:",␣
 ↪best_fitness)
```

```
Best Parameter Vector: [0.46888632 0.54627467] Best Fitness: -0.5182703869594648
Best Parameter Vector: [0.39924071 0.45875045] Best Fitness: -0.3698451271509875
Best Parameter Vector: [0.42085799 0.35316008] Best Fitness: -0.3018434925009673
Best Parameter Vector: [0.43818061 0.237306  ] Best Fitness: -0.2483163874620917
Best Parameter Vector: [0.40625321 0.32200055] Best Fitness: -0.2687260309046179
Best Parameter Vector: [0.33628493 0.28180303] Best Fitness: -0.1925005040925098
Best Parameter Vector: [0.27561629 0.2262637 ] Best Fitness:
-0.12715959984775912
Best Parameter Vector: [0.02939499 0.30611676] Best Fitness:
-0.09457153844125527
Best Parameter Vector: [0.05723213 0.13434918] Best Fitness:
```

```
-0.021325218863756108
Best Parameter Vector: [0.16193774 0.02373301] Best Fitness:
-0.026787086895781784
```

13 ES(1+1) with 1/5 rule

[ ]:
```python
def f(x):
    return x**2

# ES parameters
sigma = 0.1  # Initial mutation step size
max_generations = 100

# Success rule parameters
success_counter = 0
success_rate = 1/5  # The 1/5 rule: Acceptance rate threshold

# Initial guess for the solution
x = 5.0  # You can choose any initial value

# Main (1+1) ES loop
generations_counter = 0
for generation in range(max_generations):
    # 1. Create 1 candidate solution by applying a random mutation
    candidate = x + random.gauss(0, sigma)

    # 2. Evaluate the fitness of the candidate solution
    current_fitness = f(x)
    candidate_fitness = f(candidate)

    # 3. Decide whether to accept the candidate based on fitness
    if candidate_fitness < current_fitness:
        x = candidate  # Accept the candidate solution
        success_counter += 1
    generations_counter += 1

    # 4. Check the success rate against the 1/5 rule
    if success_counter / generations_counter >= success_rate:
        # Increase mutation step size if the success rate is met
        sigma *= 1.5
        success_counter = 0  # Reset success counter
        generations_counter = 0 # Reset generations counter
    else:
        # If not that many of the new surrounding points are better, decrease␣
 ↪sigma
        sigma *= 0.9
```

```
    # Print the best solution in this generation
    print(f"Best Solution = {x:.4f}, Best Fitness = {current_fitness:.4f},␣
↪Sigma = {sigma:.4f}")
```

14 Cov matrix

[3]:
```
x = [92, 60, 100]
y = [80, 30, 70]

normed_X = np.array([x - np.mean(x), y - np.mean(y)])

C = np.matmul(normed_X,normed_X.T) / (len(x)-1)
print(C)
```

```
[[448. 520.]
 [520. 700.]]
```