题目2

> 附注：全部作业题目源码可到 https://github.com/FRIEDparrot/intellegence_optimization_algorithm 下载和运行

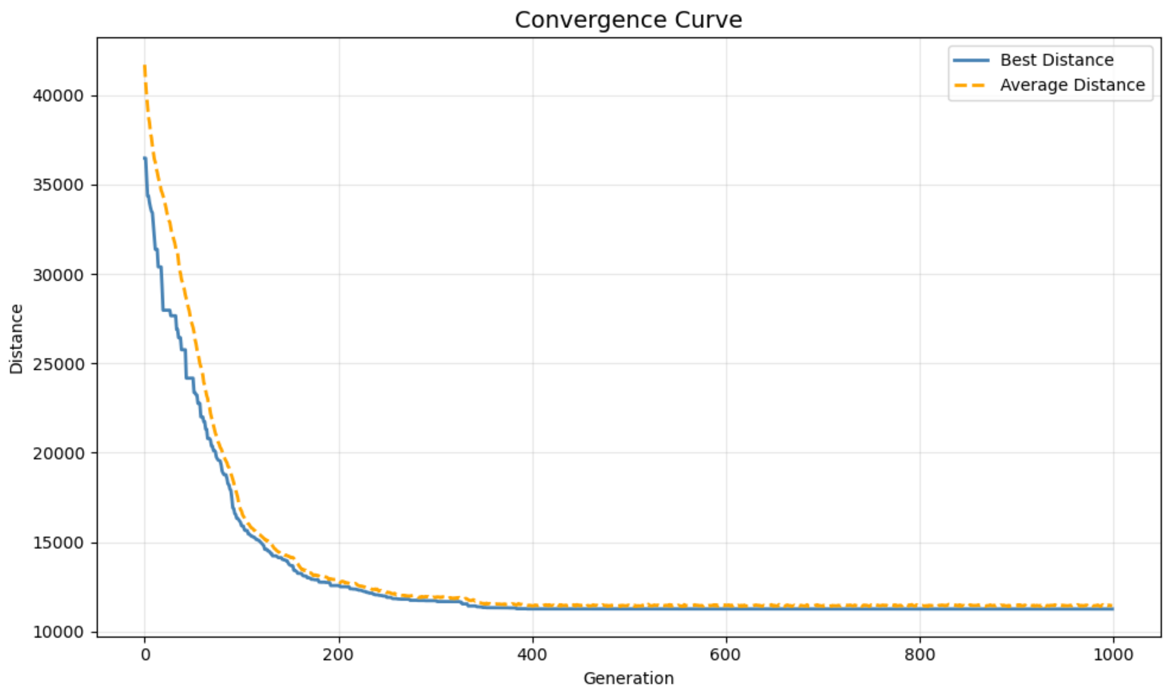
1. 不采用自适应变异， 同时不采用疫苗时：

```
best_seq = tsp(cities,
        n_population=700,
        n_generations=1000,
        mutate_rate=0.2,
        elitism_rate=0.4,
        use_immune=False,
        n_immune=10,
        immune_elite_rate=0.15,  # use first 15% as top layer
        cross_algorithm='PMX',
        mutation_algorithm='normal'
    )
```

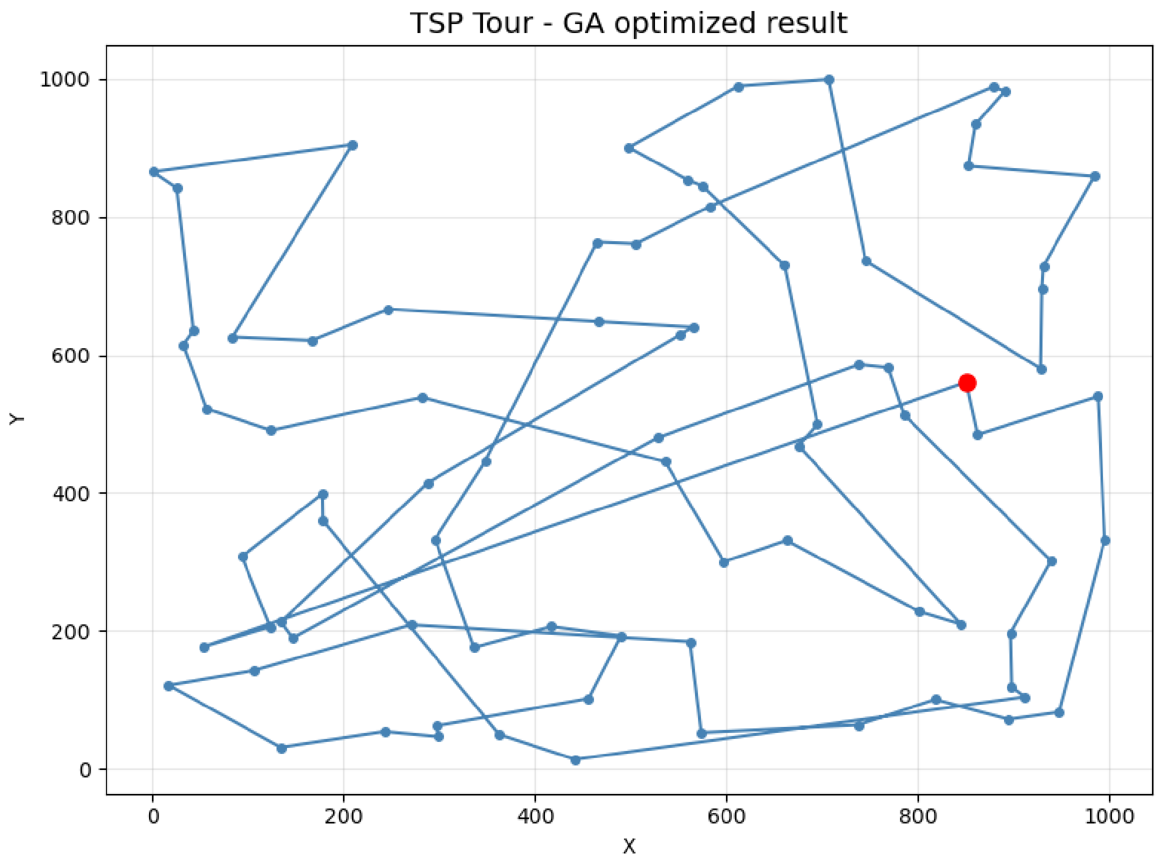
结果为：

```
Minimum distance in group:  11263.888702592736
Best sequence:  [ 0 18 55 76 35 70 59 40 38 67 32 36 17 45 71 78 77 75 23 34 62
47 51 58
 79  2  3 14 41 31 42 39  1 16 66 56  5  8 50  9 13 28 64 19  7 53 68 29
 27 30 61 21 26 49  4 63 12 37 11 48 33 20 57 69 25  6 10 15 43 46 73 54
 65 72 22 60 44 74 24 52]
```

收敛曲线如下：



最终路径如下：

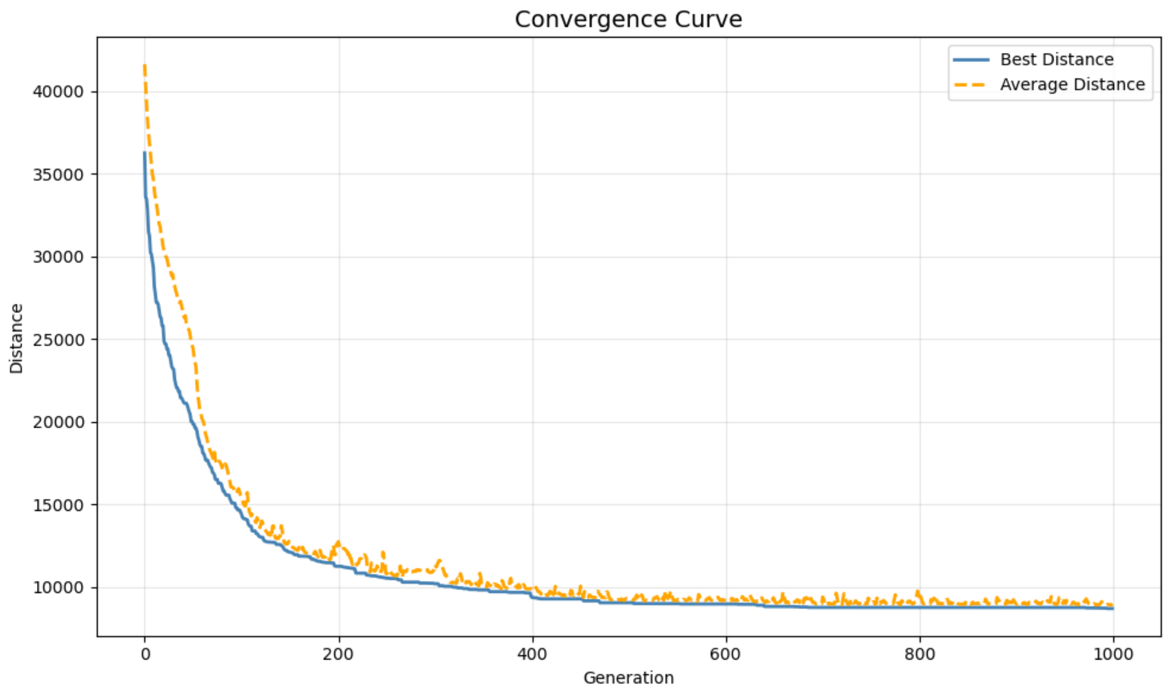


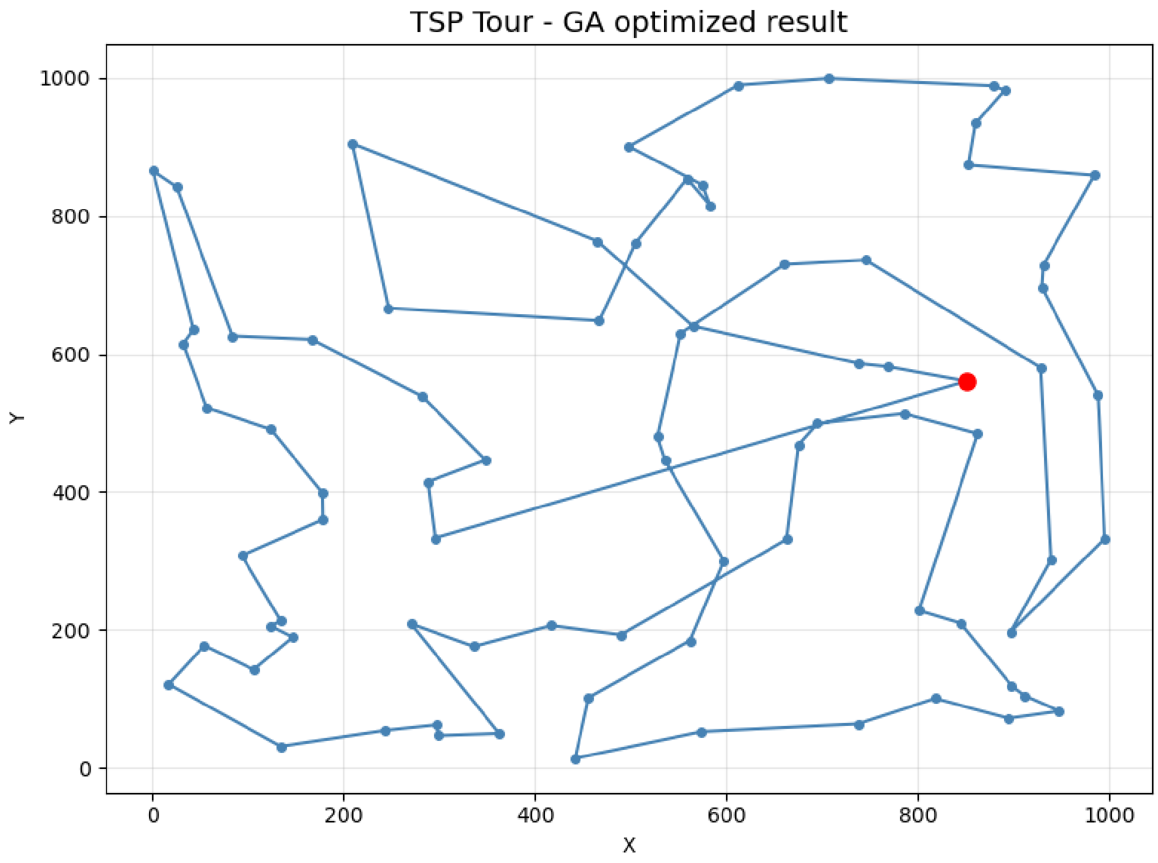
当采用分层疫苗接种时， 即仅设置 `use_immune=True` ，结果为：

```
Minimum distance in group:  8696.501031370508
Best sequence:  [ 0 15 10 33 58 63 11 48 79 50  2  9  8  5 56  3 14 41 31 42 39
 1 55 76
 73 46 16 66 13 20  6 29 68 67 75 72 38 40 59 70 35 65 54 19  7 18 43 28
 64 53 23 34 62 32 22 78 77 71 45 17 52 36 25 24 69 74 60 44 30 61 21 26
  4 49 12 37 27 51 57 47]
```

显然， 使用疫苗将最终路径长度从 11264 减少到了 8696, 显著提升了最终的收敛效果。

其收敛曲线和路径分别为：



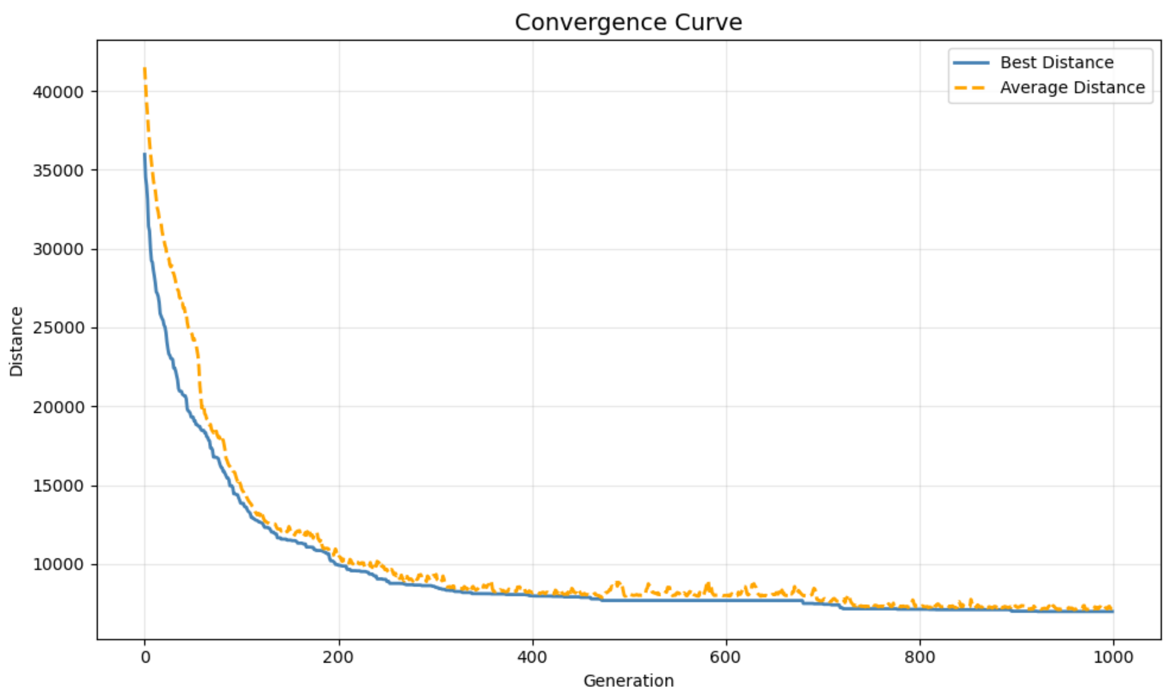


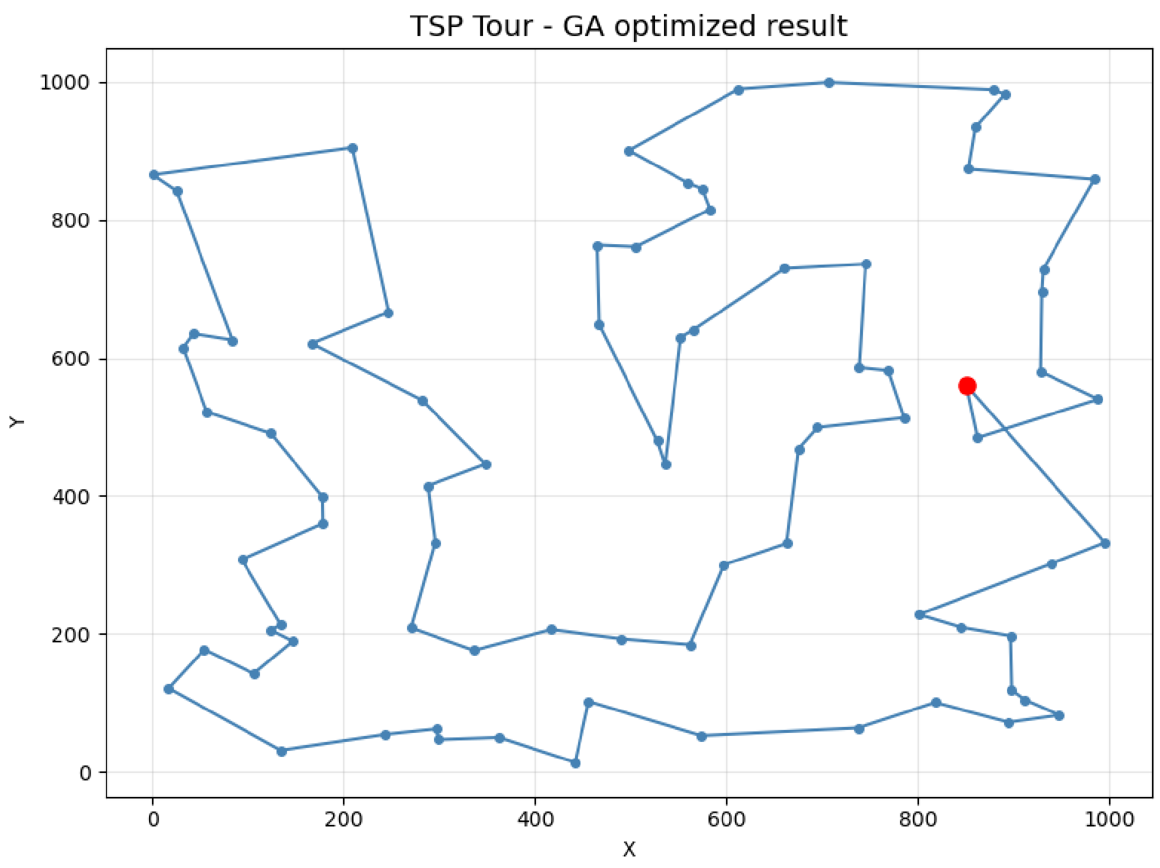
但是如果采用自适应变异办法， 则可以获取到最优的结果， 即自适应变异对于后期搜索收敛非常重要， 采用如下配置（`mutation_algorithm='adaptive'`）：

```
best_seq = tsp(cities,
        n_population=700,
        n_generations=1000,
        mutate_rate=0.2,
        elitism_rate=0.4,
        use_immune=True,
        n_immune=10,
        immune_elite_rate=0.15,  # use first 15% as top layer
        cross_algorithm='PMX',
        mutation_algorithm='adaptive'
    )
```

得到目前的最佳结果约为 6980 左右：

```
Minimum distance in group:  6980.318679212413
Best sequence:  [ 0 18 55 16  1 39 42 31 41 14  3 56  5  8 50  9  2 79 58 48  6
29 20 33
 13 66 10 15 43 28 64 53 68 67 23 34 62 32 47 57 51 27 37 11 63  4 49 12
 26 21 61 30 44 60 74 69 24 25 36 52 17 45 71 77 78 22 72 75 38 40 59 70
 35 65 54 73 19  7 46 76]
```





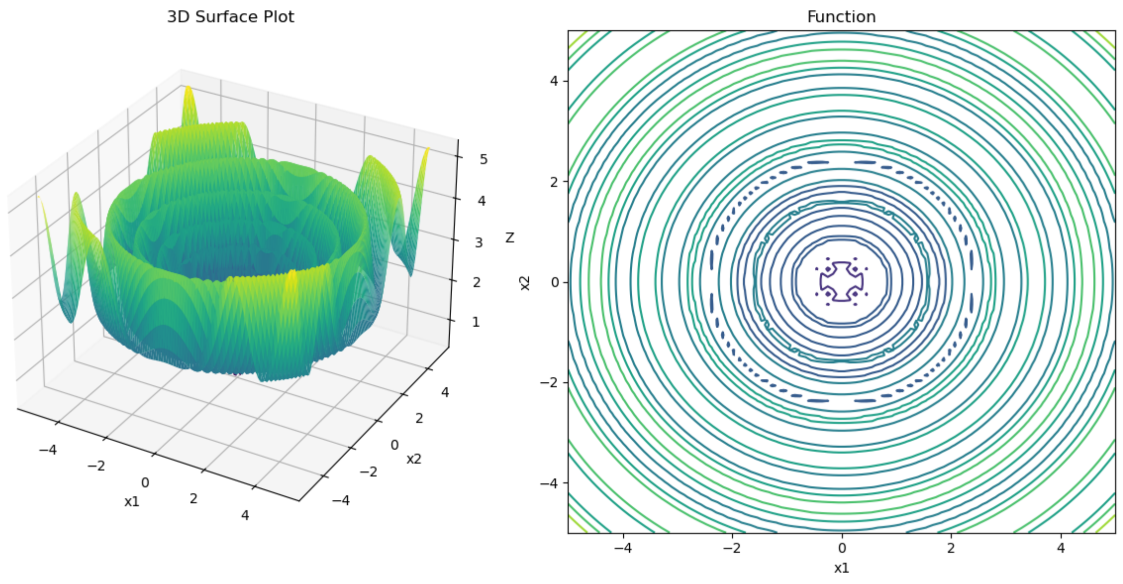
虽然上述图像应当还有更优解， 但是应当可以通过增加代数来达到， 该图结果已经为比较优化的结果。

题目3

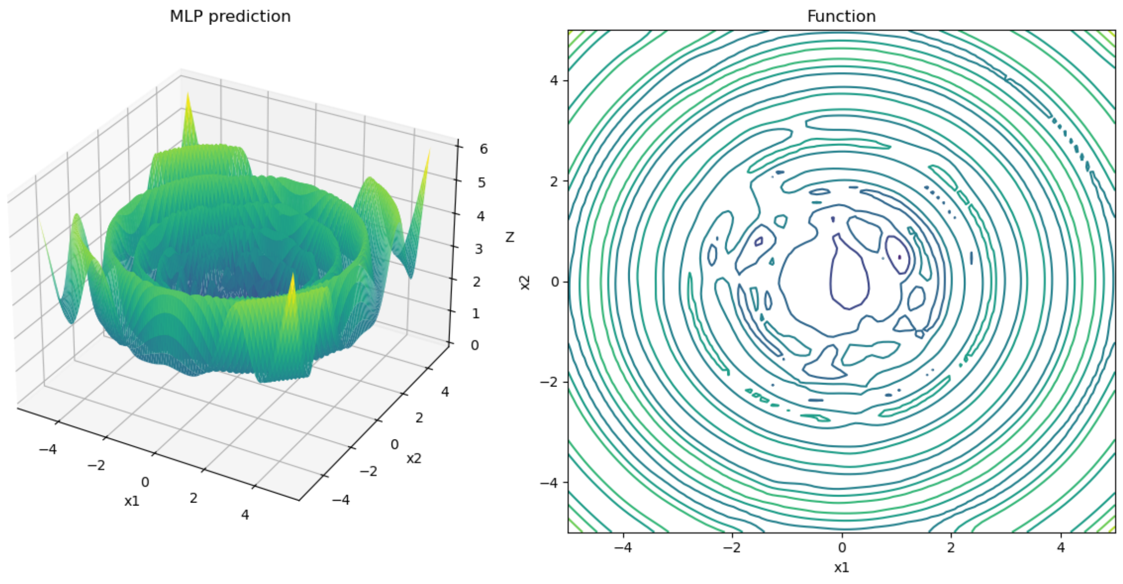
训练过程迭代输出：

```
torch.Size([2000, 2]) torch.Size([2000])
epoch:  0 loss:  0.5682784914970398
epoch:  100 loss:  0.41453665494918823
epoch:  200 loss:  0.38940805196762085
epoch:  300 loss:  0.35017839074134827
epoch:  400 loss:  0.367119699716568
epoch:  500 loss:  0.14435255527496338
epoch:  600 loss:  0.05648289620876312
epoch:  700 loss:  0.023599792271852493
epoch:  800 loss:  0.006914088502526283
epoch:  900 loss:  0.023795221000909805
Average Train loss:  0.00012471895024646074
Average Test loss:  0.00010004115029005334
```

The real function is:



the predicted function is:



题目4

采用粒子群算法进行优化求解：

参数设置如下：

- 粒子群大小：500
- 迭代次数：1000

```
best_p, best_err = pso_algorithm(yp,xp,
                                   lim_min,
                                   lim_max,
                                   inertia=0.8,
                                   c1=2,
                                   c2=2,
                                   swarm_size=500,
                                   epochs=1000)
```

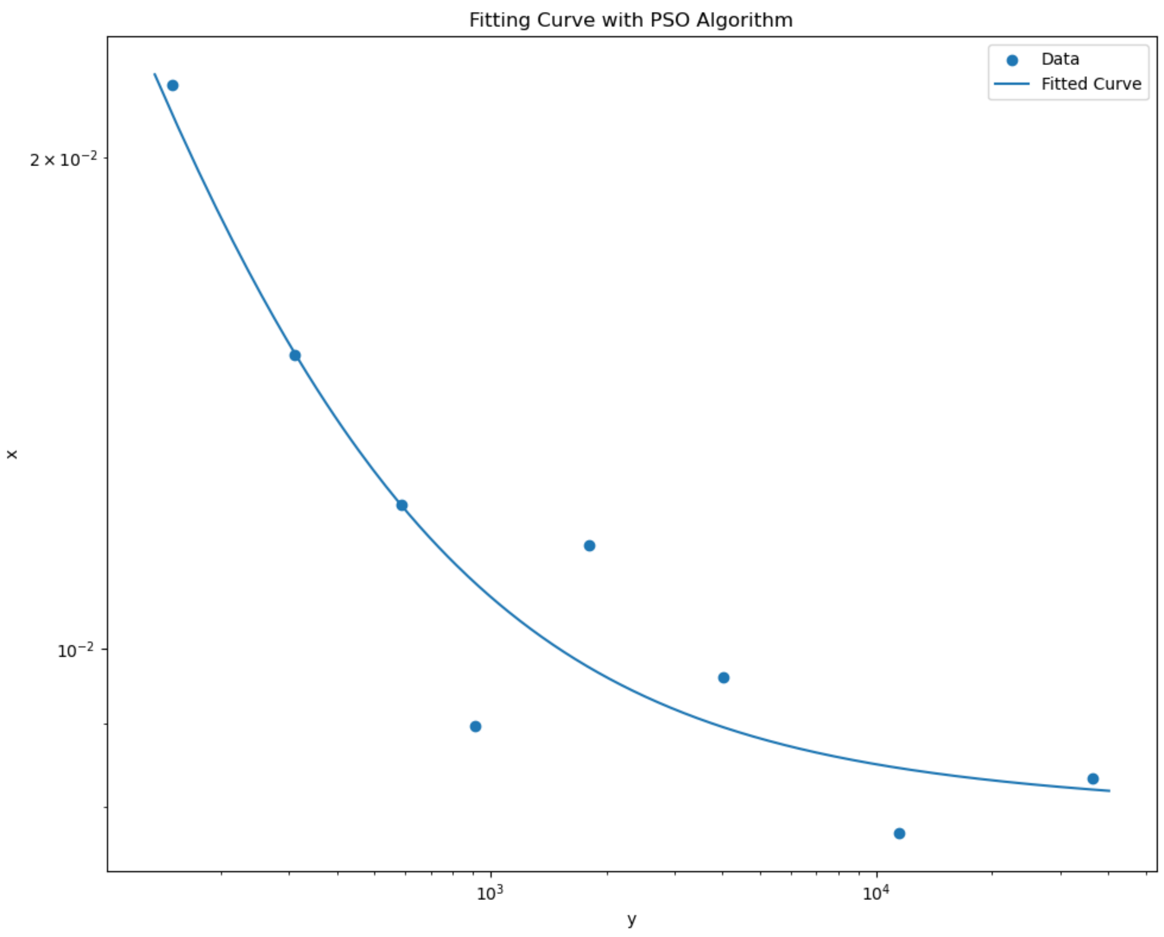
得到最终各个参数的拟合结果为：

```
Best Parameters: [ 0.0086986  -0.00692268  1.          -0.86885857]
Best Error: 0.0020 # 实际为 0.002042 左右
```

即由粒子群方法得到疲劳曲线拟合公式为：

$$x = 0.0086986 \times y^{-0.00692268} + 1 \times y^{-0.86885857} \tag{1}$$

绘制拟合曲线如下：



Appendix : Source Code of Examples

exm2

This example has 2 files, including :

1. crosslib.py :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def pmx(p1, p2):
    # select the random crossover points
    l = len(p1)
    cp1, cp2 = np.sort(np.random.choice(l, size=2, replace=False))
```

```python
    # initialize children (since we start from 0, we initialize -1 for unselected
cities)
    child1 = np.zeros(l, dtype=np.int32) - 1
    child2 = np.zeros(l, dtype=np.int32) - 1

    child1[cp1:cp2] = p2[cp1:cp2]
    child2[cp1:cp2] = p1[cp1:cp2]

    # create map1 from p1 to p2
    map1 = {}  # map from p1 to child1
    p1_ = p1[cp1:cp2]
    p2_ = p2[cp1:cp2]
    for a, b in zip(p1_, p2_):
        # we not map a non-exist element to an existing element, so continue
        #         search for non-exist mapping element in p2
        if a not in p2_:
            while b in p1_:
                b = p2_[np.where(p1_ == b)[0][0]]
            map1[a] = b  # map a to b in child1
    map2 = {b: a for a, b in map1.items()}  # reverse map

    for i in np.concatenate([np.array(range(cp1)), np.array(range(cp2,
l))]).astype(np.int32):
        child1[i] = p1[i] if p1[i] not in child1 else map2[p1[i]]
        child2[i] = p2[i] if p2[i] not in child2 else map1[p2[i]]

    return child1, child2

def ox(parent1, parent2):
    """
    Order Crossover (OX) - may be more reliable than PMX for TSP

    This part of code is generated by CHATGPT for comparing purposes only.
    """
    size = len(parent1)

    # Choose two random crossover points
    start, end = sorted(np.random.choice(size, 2, replace=False))

    # Create children
    child1 = np.full(size, -1)
    child2 = np.full(size, -1)

    # Copy segments from parents
    child1[start:end] = parent1[start:end]
    child2[start:end] = parent2[start:end]

    # Fill remaining positions
    def fill_child(child, other_parent):
        remaining = [x for x in other_parent if x not in child]
        j = 0
        for i in range(size):
            if child[i] == -1:
                child[i] = remaining[j]
                j += 1
        return child

    child1 = fill_child(child1, parent2)
    child2 = fill_child(child2, parent1)

    return child1.astype(np.int32), child2.astype(np.int32)
```

2. exm2.py, which is the main function

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from cross_lib import ox, pmx
import os

# set work dir to current file
os.chdir(os.path.dirname(os.path.abspath(__file__)))

data = pd.read_csv("data/cities.csv", header=None).transpose()
data.columns = ['x', 'y']
cities = np.array(data)

def compute_distance_matrix(cities):
    n = len(cities)
    dist_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            dist_matrix[i, j] = np.linalg.norm(cities[i] - cities[j])
    return dist_matrix

distance_matrix = compute_distance_matrix(cities)

def distance(seqs, dist_matrix):
    assert seqs.dtype == np.int32, "The sequence should be a numpy array of
int32"
    dist = np.zeros(seqs.shape[0])
    for i in range(seqs.shape[0]):
        seq = seqs[i]
        d = 0.0
        for j in range(len(seq) - 1):
            d += dist_matrix[seq[j], seq[j + 1]]  # 直接查距离方阵
        dist[i] = d
    return dist

def norm_path(path: np.ndarray):
    """
    This function is used to check if a path is equivalent to its reverse or
cyclic permutation.

    1. Check the first city and permute to equal path.
    2. Check the second city number and permute to equal path.
    :param path:
    :return:
```

```
    """
    assert path.ndim == 1, "The path should be a 1D array"
    assert 0 in path, "The path should start from the first city (0)"  # check
if the path starts from the first city
    path = path.copy()
    # Find index of city 0
    start_idx = np.where(path == 0)[0][0]
    # Rotate to start from city 0
    path = np.roll(path, -start_idx)

    # Choose direction (smaller second city)
    if path[1] > path[-1]:
        path[1:] = path[1:][::-1]
    return path

def normal_mutation(individual, rate):
    """
    This function is used to mutate the sequence by swapping two cities.

    Implemented by me firstly and fixed as normal mutation function
    :param individual:
    :param rate:
    :return:
    """
    if np.random.rand() < rate:
        i,j = np.random.choice(len(individual), size=2, replace=False)
        individual[j], individual[i] = individual[i], individual[j]   # swap two
random elements
    individual = norm_path(individual)  # make sure the path is valid
    return individual

def adaptive_mutation(individual, mutation_rate):
    """
    Adaptive mutation that decreases over time and uses different strategies

    This function is generated by ChatGPT for optimizing the mutation process,
        and exploiting

    It is tested that using adaptive mutation can improve the performance
greatly

    :param individual:
    :param mutation_rate:
    """
    if np.random.random() < mutation_rate:
        strategy = np.random.choice(['swap', 'insert', 'reverse'])

        if strategy == 'swap':
            # Swap two random cities
            i, j = np.random.choice(len(individual), 2, replace=False)
            individual[i], individual[j] = individual[j], individual[i]

        elif strategy == 'insert':
            # Remove a city and insert it elsewhere
            i = np.random.randint(len(individual))
```

```python
            j = np.random.randint(len(individual))
            city = individual[i]
            individual = np.delete(individual, i)
            individual = np.insert(individual, j, city)

        elif strategy == 'reverse':
            # Reverse a segment
            i, j = sorted(np.random.choice(len(individual), 2, replace=False))
            individual[i:j + 1] = individual[i:j + 1][::-1]

    return individual

def cross(parent_genes,
          child_number,
          mutation_rate=0.1,
          algorithm='PMX',
          mutation_algorithm='normal'):
    """
    In The example MATLAB code, the crossover method is simple Order Crossover
(OX).
    Here, we introduce the partial Mapped Crossover (PMX) method.

    :param parent_genes:
    :param child_number:
    :param mutation_rate: the rate of mutation
    :param algorithm: 'PMX' or 'OX'
    :param mutation_algorithm: 'normal' or 'adaptive'
    :return:
    """
    assert child_number % 2 == 0, "The number of children should be even"
    n, l = parent_genes.shape   # number of parents

    children = []
    for _ in range(child_number // 2):
        p1_idx, p2_idx = np.random.choice(n, size=2, replace=False)
        p1, p2 = parent_genes[p1_idx].copy(), parent_genes[p2_idx].copy()

        if algorithm == 'PMX':
            child1, child2 = pmx(p1, p2)
        elif algorithm == 'OX':
            child1, child2 = ox(p1, p2)
        else:
            raise NotImplementedError

        children.append(norm_path(child1))
        children.append(norm_path(child2))

    new_genes = np.vstack(children)

    # ----- mutation -----
    n, l = new_genes.shape
    for i in range(n):
        if mutation_algorithm == 'normal':
            new_genes[i,:] = normal_mutation(new_genes[i, :], mutation_rate)
        elif mutation_algorithm == 'adaptive':
            new_genes[i, :] = adaptive_mutation(new_genes[i,:], mutation_rate)
        else:
            raise NotImplementedError

        new_genes[i,:] = norm_path(new_genes[i,:])
    new_genes = np.vstack([parent_genes, new_genes])
    return new_genes

def generate_vaccines(path_len, n_immune=5):
    """
    This function is used to generate unique vaccine pairs (city swap positions)
    :param path_len: the length of the path
    :param n_immune: the number of vaccines to generate
    :return:
    """
    vaccines = []
    while len(vaccines) < n_immune:
        # sample a vaccine
        new_vaccine = tuple(sorted(np.random.choice(path_len, size=2,
replace=False)))
        if new_vaccine not in vaccines:
            vaccines.append(new_vaccine)
    return np.array(vaccines)

def apply_vaccine(seq, vaccine):
    """Apply vaccine (swap two cities) to a sequence"""
    new_seq = seq.copy()
    m, n = vaccine
    new_seq[m], new_seq[n] = new_seq[n], new_seq[m]
    return new_seq

def immune_process(seqs, n_immune=5, elitism_rate = 0.2):
    """
    The immune system for TSP

    We apply the following thought for better immune system :
    - sample for `n_immune` types of vaccines
    - firstly split the group into elitism and non-elitism group
    - for elitism group, we only apply the vaccine that most suit for the
individual
    - for non-elitism group, we apply the vaccine that most suit for this group

    :param seqs:
    :param n_immune:
    :param elitism_rate:
    :return:
    """
    nums, seq_len = seqs.shape   # s : number of individuals, l : length of the
path
    vaccines = generate_vaccines(seq_len, n_immune)  # generate vaccine

    # we sample for 5 types of vaccines to test on the population
    d0 = distance(seqs, distance_matrix)
    elite_num  = int(nums * elitism_rate)
    top_layer = np.argsort(d0)[:elite_num]  # select the top individuals
(minimum distance)
    non_top_layer = np.delete(np.arange(nums), top_layer)  # select the non-top
individuals from the population

    # --- Elite Layer vaccination ---
    for idx in top_layer:
        vacc_dist = []
        elite_seqs = seqs[idx, :].copy()   # make a copy of the individual
        for vaccine in vaccines:
            new_seq = apply_vaccine(elite_seqs, vaccine)  # apply the vaccine to
the individual
            d_new = distance(new_seq.reshape(1, -1), distance_matrix)[0]  #
calculate the distance
            vacc_dist.append(d_new)

        best_vaccine_idx = np.argmin(vacc_dist)  # find the best vaccine idx
        if vacc_dist[best_vaccine_idx] < d0[idx]:  # best vaccine is good for
this individual
            # apply the vaccine to the individual
            seqs[idx,:] =  apply_vaccine(seqs[idx,:],
vaccines[best_vaccine_idx])

    #  --- Non-Elite Layer vaccination ---
    if len(non_top_layer) > 0:  # if there are non-top individuals
        vacc_dist = []
        for vaccine in vaccines:
            m, n = vaccine
            group_copy = seqs[non_top_layer].copy()
            group_copy[:, m], group_copy[:, n] = group_copy[:, n], group_copy[:,
m]
            avg_dist = np.mean(distance(group_copy, distance_matrix))
            vacc_dist.append(avg_dist)     # calculate the mean distance of the
non-top  group

        best_vaccine_idx = np.argmin(vacc_dist)  # find the best vaccine idx
        # apply the best vaccine to the non-top group
        current_avg_dist = np.mean(d0[non_top_layer])
        if np.min(vacc_dist) < current_avg_dist:
            m, n = vaccines[best_vaccine_idx]
            # vaccine to the group
            seqs[non_top_layer, m], seqs[non_top_layer, n] = seqs[non_top_layer,
n], seqs[non_top_layer, m]
    return seqs

def tsp(cities: np.ndarray,
        n_population = 300,
        n_generations = 100,
        mutate_rate = 0.25,
        elitism_rate = 0.5,
        use_immune = False,
        n_immune = 5,
        immune_elite_rate = 0.2,
        cross_algorithm='PMX',
        mutation_algorithm='adaptive') :
```

```
    """
    Traveling Salesman Problem (TSP) using GA (Genetic Algorithm)
    :param cities:
    :param n_population:
    :param n_generations:
    :param mutate_rate:
    :param elitism_rate:
    :param use_immune:
    :param n_immune: see immune_process function for details
    :param immune_elite_rate: see immune_process function for details
    :param cross_algorithm: see cross function for details
    :param mutation_algorithm: see cross function for details
    :return:
    """

    # initialize the population
    seqs = np.vstack([np.random.permutation(len(cities))
                      for _ in range(n_population)]).astype(np.int32)
    seqs = np.vstack([norm_path(seqs[i, :]) for i in range(n_population)])
    d = distance(seqs, distance_matrix)
    min_dist = 1e10
    best_seq = None

    min_dist_arr = []
    avg_dist_arr = []
    # use GA algorithm to optimize the TSP problem
    for epoch in range (n_generations):
        """
        we use max(d) - d as the fitness function to make it minimization
        The higher the fitness, the higher probability of the individual
            being selected as the parent of the next generation.
        """
        fitness = (np.max(d) - d) / (np.max(d) - np.min(d) + 1e-10)   #
normalize the fitness to [0,1]
        prob = fitness / np.sum(fitness)  # not plus 1e-10 for sum prob to be 1
        # select the parents using roulette wheel selection

        # only keep the best individuals for the next generation
        n_parents = int(n_population * elitism_rate)   # number of parents to
keep
        parents = np.argsort(prob)[-n_parents:]   # select the top n_parents
individuals

        # generate children using crossover
        parent_genes = seqs[parents,:]
        child_number = n_population - len(parents)

        seqs = cross(parent_genes,
                     child_number,
                     mutation_rate=mutate_rate,
                     algorithm=cross_algorithm, # use PMX crossover by default
                     mutation_algorithm=mutation_algorithm)

        # use immune system to avoid getting stuck in local optima
        if use_immune:
            seqs = immune_process(seqs, n_immune, immune_elite_rate)

        d = distance(seqs, distance_matrix)   # update the distances of the new
population
        ### evaluate process ###
        best_dist = np.min(d)
        avg_dist  = np.mean(d)

        if best_dist < min_dist:
            min_dist = best_dist
            best_seq = seqs[np.argmin(d)]
            draw_path(cities, best_seq, title="TSP Tour - Epoch: " + str(epoch))

        min_dist_arr.append(min_dist)
        avg_dist_arr.append(avg_dist)
        # note : high mutation rate can be harmful for convergence so we not use
it.
        print("Epoch: ", epoch, " Best distance in group: ",
              best_dist, "Minimum distance: " , min_dist)
    print("Minimum distance in group: ", np.min(d))
    # plot the convergence image
    draw_convergence(min_dist_arr, avg_dist_arr)
    return best_seq

def draw_path(cities, path, title="TSP Tour"):
    """
    Plot the path of the TSP tour.

    Generated by ChatGPT
    :param cities: coordinates (n,2)
    :param path: ordered city sequence
    :param title:
    """
    plt.figure(figsize=(8,6))
    # path, closing the tour
    tour = np.append(path, path[0])
    plt.plot(cities[tour, 0], cities[tour, 1], '-o', color='steelblue',
markersize=4, linewidth=1.5)

    # highlight starting city
    plt.plot(cities[path[0], 0], cities[path[0], 1], 'ro', markersize=8)

    plt.title(title, fontsize=14)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.show()


def draw_convergence(history_min_dist, history_avg_dist, title="Convergence
Curve"):
    """
    Plot the convergence curve of the TSP optimization process.
    Args:
        history_min_dist (list): List of minimum distances per generation.
        history_avg_dist (list): List of average distances per generation.
        title (str): Title of the plot.

    Generated by ChatGPT
    """
    plt.figure(figsize=(10, 6))
    generations = range(len(history_min_dist))

    # Plot minimum and average distance
    plt.plot(generations, history_min_dist, label="Best Distance",
color="steelblue", linewidth=2)
    plt.plot(generations, history_avg_dist, label="Average Distance",
color="orange", linestyle="--", linewidth=2)

    plt.xlabel("Generation")
    plt.ylabel("Distance")
    plt.title(title, fontsize=14)
    plt.grid(alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.show()

def main():
    best_seq = tsp(cities,
                   n_population=700,
                   n_generations=1000,
                   mutate_rate=0.2,
                   elitism_rate=0.4,
                   use_immune=True,
                   n_immune=10,
                   immune_elite_rate=0.15,  # use first 15% as top layer
                   cross_algorithm='PMX',
                   mutation_algorithm='adaptive'
                   )

    print("Best sequence: ", best_seq)
    draw_path(cities, best_seq, title="TSP Tour - GA optimized result")   # plot
the best sequence

if __name__ == "__main__":
    main()

def test_pmx():
    num = 200
    p1 = np.random.permutation(np.arange(num))
    p2 = np.random.permutation(np.arange(num))
    res = cross(np.vstack([p1, p2]), 2)
    assert np.all(sorted(res[2]) == np.arange(len(p1)))
    assert np.all(sorted(res[3]) == np.arange(len(p2)))

def test_immune_process():
    num = 20  # use 20 cities for test
    l = 30    # length of the path
    seqs = np.vstack([np.random.permutation(l) for _ in
range(num)]).astype(np.int32)
    d = distance(seqs, distance_matrix)
    seqs = immune_process(seqs, n_immune=5, elitism_rate=0.2)
    d2 = distance(seqs, distance_matrix)
    print("Distance before immune process: ", d)
    print("Distance after immune process: ", d2)

    assert(np.min(d2) <= np.min(d))
```

exm3

```
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split

def func(x):
    return (x[:, 0]**2 + x[:, 1]**2) ** 0.25 * (torch.sin(50 * (x[:, 0]**2 +
x[:, 1]**2) ** 0.1) **2 + 1)


def plot_function(func=None, title=None, is_model=False):
    # generate data
    x1_lim = [-5, 5]
    x2_lim = [-5, 5]

    x1 = np.linspace(x1_lim[0], x1_lim[1], 100).astype(np.float32)
    x2 = np.linspace(x2_lim[0], x2_lim[1], 100).astype(np.float32)

    X1, X2 = np.meshgrid(x1, x2)
    X = np.stack([X1, X2], axis=-1)  # Changed axis to -1 to get correct shape

    # Handle both function and model cases
    if is_model:
        with torch.no_grad():
            X_tensor = torch.tensor(X.reshape(-1, 2))  # Flatten for model input
            Z = func(X_tensor).detach().numpy()
            Z = Z.reshape(100, 100)  # Reshape back to grid
    else:
        Z = func(torch.tensor(X.reshape(-1, 2))).detach().cpu().numpy()
        Z = Z.reshape(100, 100)  # Reshape back to grid

    # plot the function
    fig = plt.figure(figsize=(12, 6))
    ax1 = fig.add_subplot(121, projection='3d')
    surf = ax1.plot_surface(X1, X2, Z, rstride=1, cstride=1, cmap='viridis',
edgecolor='none')
    ax1.set_xlabel('x1')
    ax1.set_ylabel('x2')
    ax1.set_zlabel('Z')
    ax1.set_title(title)

    ax2 = fig.add_subplot(122)
    contour = ax2.contour(X1, X2, Z)
    ax2.set_xlabel('x1')
    ax2.set_ylabel('x2')
    ax2.set_title('Function' if not is_model else 'Prediction')
    plt.tight_layout()
    plt.show()

class SimpleMLP(nn.Module):
    def __init__(self, hidden_size=1024, lr=0.01):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(2, hidden_size)
        self.gelu1 = nn.GELU()
        self.fc2 = nn.Linear(hidden_size, hidden_size )
        self.gelu2 = nn.GELU()
        self.fc3 = nn.Linear(hidden_size, hidden_size )
        self.gelu3 = nn.GELU()
        self.fc4 = nn.Linear(hidden_size, 1)

        self.net = nn.Sequential(
            self.fc1,
            self.gelu1,
            self.fc2,
            self.gelu2,
            self.fc3,
            self.gelu3,
            self.fc4
        )
        self.optim = torch.optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        self.to(self.device)  # move the model to GPU if available

    def forward(self, x: torch.Tensor):
        x = x.to(self.device)
        return self.net(x)

    def train_model(self, X, y, epochs=1000, batch_size=32):
        """
        Use SGD to train the model
        """
        X = X.to(self.device)
        y = y.to(self.device).unsqueeze(1)  # add a dimension for the output

        dataloader = DataLoader(list(zip(X, y)), batch_size=batch_size,
shuffle=True)
        self.train()
        for epoch in range(epochs):
            for X, y in dataloader:
                self.optim.zero_grad()
                y_pred = self.forward(X)
                loss = self.loss(y_pred, y)
                loss.backward()
                self.optim.step()

            if epoch % 100 == 0:
                print( "epoch: ", epoch, "loss: ", loss.item())
                plot_function(func=self, title=f'MLP epoch {epoch}')

        # calculate the train loss
        self.eval()
        y_pred = self.predict(X)
        loss = self.loss(y_pred, y) / len(y)
        print("Average Train loss: ", loss.item())

    def test_model(self, X, y):
        """
        Test the model on the test set
        """
        self.eval()
        X = X.to(self.device)
        y = y.to(self.device).unsqueeze(1)

        y_pred = self.predict(X)
        loss = self.loss(y_pred, y) / len(y)
        print("Average Test loss: ", loss.item())

    def predict(self, X):
        return self.forward(X)


def main():
    mlp = SimpleMLP(lr=1e-4, hidden_size=1024)

    # generate 2000 data points ()
    x = torch.tensor(np.random.uniform(-5, 5, size=(2000,
2)).astype(np.float32))
    y = func(x)
    print(x.shape, y.shape)

    # split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=42)
    mlp.train_model(X_train, y_train, epochs=1000, batch_size=32)
    mlp.test_model(X_test, y_test)

    # plot the trained model
    plot_function(func=mlp, title='MLP prediction')
    plot_function(func=func, title='True Function')

if __name__ == '__main__':
    main()
```

exm4

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error as mse
import os

os.chdir(os.path.dirname(os.path.abspath(__file__)))

def func(y, params):
    assert params.ndim == 1 and params.shape[0] == 4
    x = (params[0] * y ** params[1]) + (params[2] * y ** params[3])
    return x

def pso_algorithm(y, x,
                  limit_min : np.array,
                  limit_max : np.array,
                  inertia = 0.8,
                  c1: float = 2,
                  c2: float = 2,
                  swarm_size = 200,
                  epochs = 1000):
    """
    Particle swarm optimization algorithm for fitting a function.
    """
    # Initialize the swarm and its velocity
    swarms = np.random.rand(swarm_size, 4) * (limit_max - limit_min) + limit_min

    v = (np.random.rand(swarm_size, 4)  - 0.5) * (limit_max - limit_min) * 0.01

    p = np.zeros_like(swarms)   # record best solution for each swarm member
    pv = np.ones(swarm_size) * 1e10  #  minimum error  of the swarm members

    best_p = np.zeros(4)       # record the best solution of the swarm
    best_err = 1e10            # record the best error of the swarm

    err_arr = np.ones(swarm_size) * 1e10  # record the error of the swarm
members

    # Calculate the initial error of the swarm members
    for epoch in range(epochs):
        for i in range(swarm_size) :
            # firstly, calculate prediction of x for each swarm member
            member = swarms[i,:]
            x_pred = func(y, member)  # predict the value of x
            x_pred = np.clip(x_pred, 1e-5 * np.ones_like(x_pred), np.inf)
            err = mse(np.log10(x), np.log10(x_pred))

            err_arr[i] = err   # record the error of the swarm member

            # update the best solution point
            if err < pv[i]:
                pv[i] = err
                p[i,:] = member  # update the best solution point
        if np.min(pv) < best_err:
            best_err = np.min(pv)  # update the best error
            best_p = p[np.argmin(pv),:]  # update the best solution

        if epoch < epochs - 1:
            # update the swarm velocity and position
            for j in range(swarm_size) :
                # Start the optimization process
                v[j, :] = (inertia * v[j,:]
                           + c1 * np.random.rand() * (p[j, :] - swarms[j, :])
                           + c2 * np.random.rand() * (best_p - swarms[j, :]))
                swarms[j,:] = v[j,:] + swarms[j,:]  # update the swarm position

            # clip the swarm position to the limit
            swarms = np.clip(swarms, limit_min, limit_max)  # limit the swarm
position

        if epoch % 100 == 0:
            print("-" * 50)
            print(f"Epoch {epoch}, Average Error: {np.mean(err_arr):.6f}, Best
Error in Group: {np.min(err_arr):.6f}")
            print(f"Best Error so far: {best_err:.6f}, Best Parameters:
{best_p}")

    return best_p, best_err

def main():
    # Reading the data from the csv file
    data = pd.read_csv('./data.csv', header=None)
    xp = np.array(data[0])
    yp = np.array(data[1])

    lim_min = np.array([0, -2, 0, -5])
    lim_max = np.array([0.1, 0, 1, 0])
    best_p, best_err = pso_algorithm(yp,xp,
                                     lim_min,
                                     lim_max,
                                     inertia=0.8,
                                     c1=2,
                                     c2=2,
                                     swarm_size=500,
                                     epochs=1000)
    print("=" * 50)  # print the best parameters and error
    print(f"Best Parameters: {best_p}, Best Error: {best_err:.4f}")

    y_test = np.linspace(0.9 * min(yp), 1.1 * max(yp), 2000)
    x_test = func(y_test, best_p)  # predict the value of x

    plt.figure(figsize=(10, 8))
    plt.scatter(yp, xp, label='Data')
    plt.plot(y_test, x_test, label='Fitted Curve')
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('y')
    plt.ylabel('x')
    plt.legend()
    plt.title(f"Fitting Curve with PSO Algorithm")
    plt.tight_layout()
    plt.show()


if __name__ == '__main__':
    main()
```