Optimisation Algorithms for Data Analysis Week 8 Assignment

Jiaming Deng 22302794

(a)

(i) Firstly, the input parameters of the global random search algorithm are the function, the amount of data, the dataset, and the number of iterations. The algorithm is implemented by first reading the data of the dataset into the l and u arrays, where l stores the minimum value and u stores the maximum value. The best x and f values are initialised as X, and F as 0 and the largest int type value respectively. Next it iterate through itr_times several times, each time taking a random number from the range of l and u and putting it into the array, and calculating the function value, updating the best x and f values if the function value is smaller than the best value.

```
def random_search(f, n, data_range, N):
    l = []
    u = []
    for Range in data_range:
        l.append(Range[0])
    for Range in data_range:
        u.append(Range[1])
    X = 0
    F = sys.maxsize
    x_list = []
    f_list = []
    for i in range(N):
        data_x = []
        for j in range(n):
            x = uniform(l[j], u[j])
            data_x.append(x)
        data_f = f(data_x[0], data_x[1])
        if data_f < F:
            X = deepcopy(data_x)
            F = data_f
        x_list.append(deepcopy(X))
        f_list.append(F)
    return x_list, f_list
```

(ii) Firstly the two functions used in this problem are as follows.

$$f_1(x, y) = 3(x-3)^4 + 9(y-8)^2$$
$$f_2(x, y) = \mathrm{Max}(x-3, 0) + 9 \cdot |y-8|$$

I use sympy to initialise the two variables x and y, define two equations based on the above functions and use the diff() function to get the partial derivatives of the two functions.

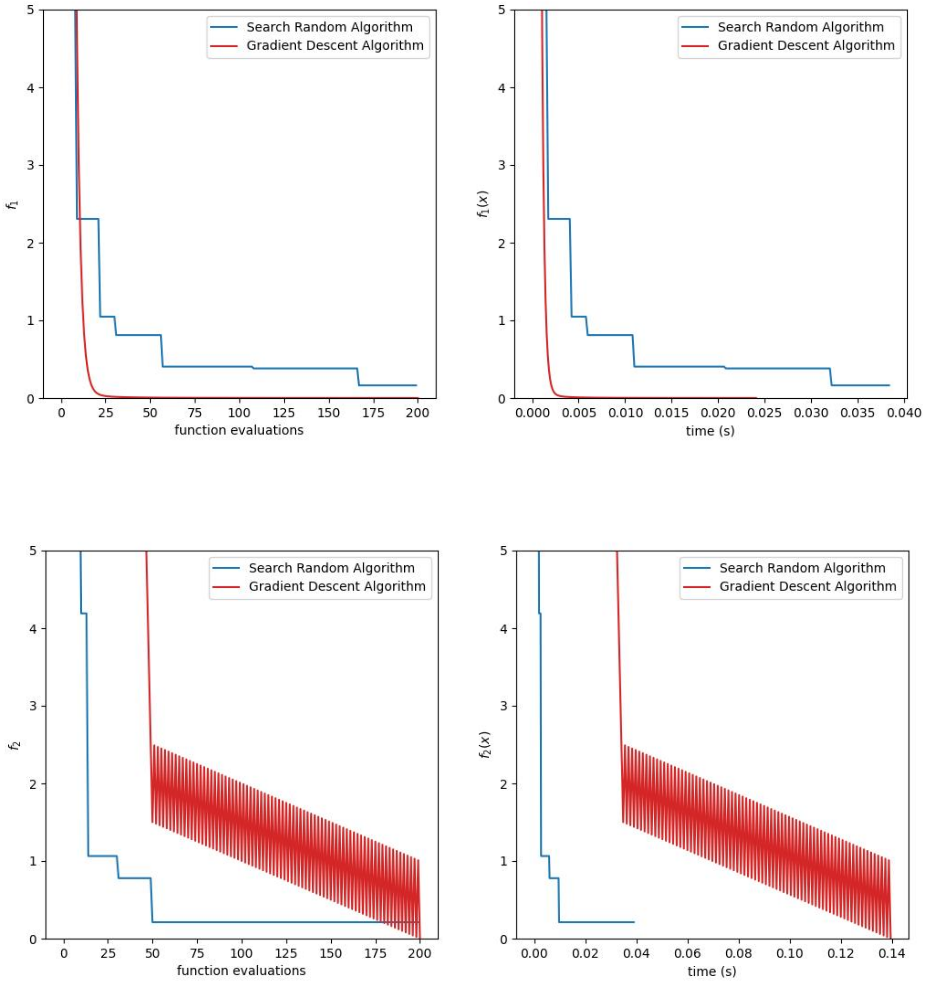
$$\frac{\partial f_1}{\partial x} = 12(x-3)^3,\ \frac{\partial f_1}{\partial y} = 18y - 144$$

$$\frac{\partial f_2}{\partial x} = \theta(x-3),\ \frac{\partial f_2}{\partial y} = 9 \cdot \mathrm{sign}(y-2)$$

The gradient descent algorithm used in this problem was implemented in the Week 2 assignment and uses a parameter alpha of 0.01. Measuring the execution time and comparing the gradient descent algorithm with the global random search algorithm is not easy because the execution times of the two algorithms are very short and the differences are small. I therefore used the timeit function, which specifies the number of times the function is executed and counts the time, and I specified that each algorithm is executed 500 times. The results of the experiment are as follows.

	Gradient Descent (s)	Global Random Search (s)
f1	0.1371	0.1743
f2	0.6604	0.1701

Next I plot the change in function value with the number of function evaluations and the change in function value with time for $f_1(x)$ and $f_2(x)$ respectively. For $f_1(x)$, the Gradient Descent algorithm converged to a minimum around the 30th iteration in about 0.005s, but the Global Search algorithm did not converge to a minimum even after 200 iterations. For $f_2(x)$, both the Gradient Descent algorithm and the global search algorithm did not converge to a minimum in 200 iterations. The Gradient Descent algorithm behaves as a jagged structure with poor stability after the 50th iteration. The Global Search algorithm converged quickly in a few iterations, but mostly showed no convergence because of the instability of the random numbers generated in the algorithm.



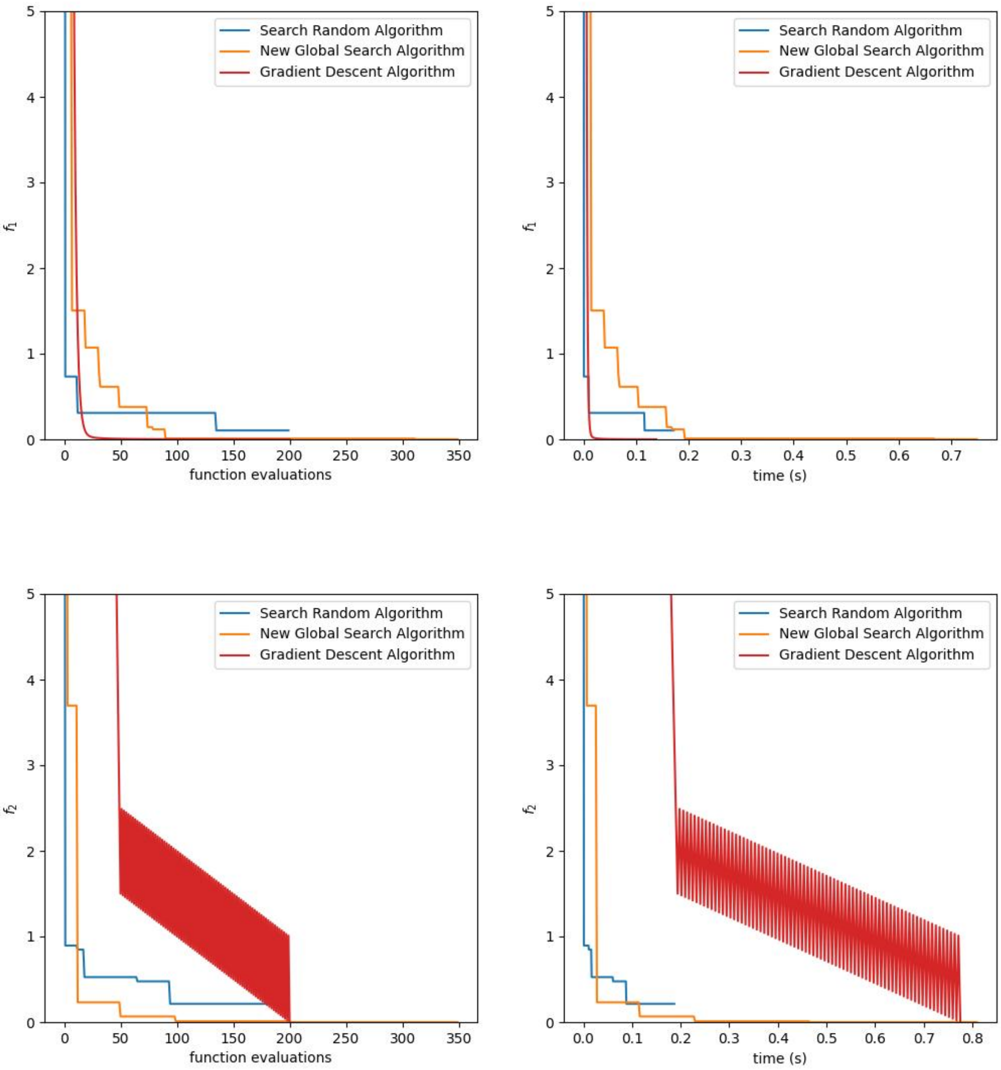
(b)

(i) The algorithm in this problem is based on the Global Random Search algorithm with some improvements. First, the minimum and maximum values of the generated numbers are specified according to the arrays l and u and an array of length N is randomly generated, the elements of the array are vectors of length 2 which are the x and y parameters of the function and the function value is calculated. Next, in line with the Global Random Search algorithm, if the function value is smaller than the previously saved minimum function value then the minimum function value is updated. Finally, in order to select M results, it is updated to N-M/M values by multiplying by a random number between 0.7 and 1.3.

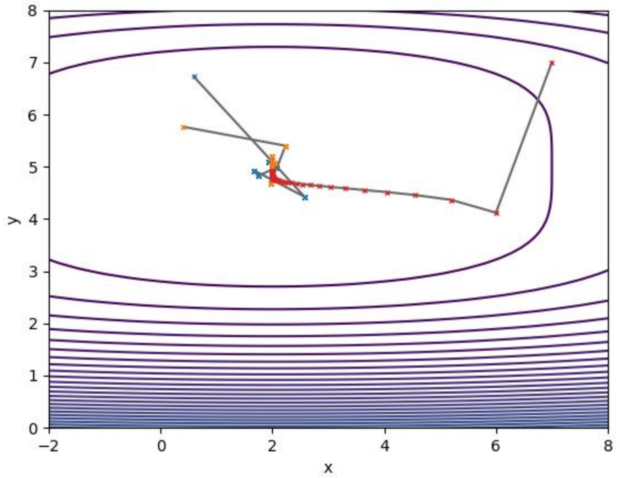
```
X = 0
F = sys.maxsize
x_list = []
f_list = []
current_x_list = []
for i in range(N):
    tmp = []
    for j in range(n):
        tmp.append(uniform(l[j], u[j]))
    current_x_list.append(tmp)
current_f_list = [0] * N
for i in range(N):
    current_x = current_x_list[i]
    current_f = f(current_x[0], current_x[1])
    current_f_list[i] = current_f
    if current_f < F:
        X = current_x
        F = current_f
    x_list.append(X)
    f_list.append(F)
opt_times = (N - M) // M
for _ in range(itr_times):
    current_f_list, current_x_list = sort_lists(current_f_list, current_x_list)
    for i in range(M):
        current_x = current_x_list[i]
        for j in range(opt_times):
            x_plus = [x * uniform(0.7, 1.3) for x in current_x]
            k = M + (i * opt_times) + j
            current_x_list[k] = x_plus
            current_f_list[k] = f(x_plus[0], x_plus[1])
            if current_f_list[k] < F:
                X = current_x_list[k]
                F = current_f_list[k]
            x_list.append(X)
            f_list.append(F)
return x_list, f_list
```

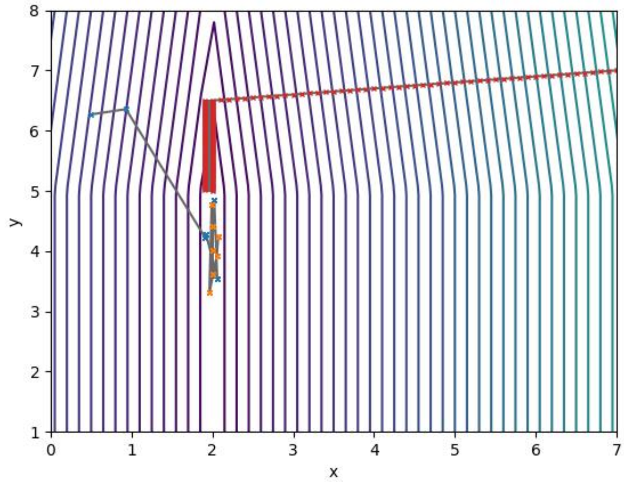
(ii) Firstly, I set the parameters N to 50, M to 20, and the number of iterations to 15 for the algorithm after improving Global Random Search, and similarly to question (a) use the timeit function to measure the running time of the algorithm and plot the change in function value with the number of function evaluations and the change in function value with time for $f_1(x)$ and $f_2(x)$ respectively and plot the change in x for $f_1(x)$ and $f_2(x)$ respectively. The results obtained are shown in the figure below.

As can be seen in the figure below, for $f_1(x)$, only the random search algorithm did not converge to a minimum during the iterations, while the other two algorithms did. The Gradient Descent algorithm reached a minimum at the 50th iteration and the New Global Search algorithm dropped to a minimum at the 100th iteration. For $f_2(x)$, only the New Global Search algorithm converged to a minimum at the 100th iteration, while the other two algorithms did not converge to a minimum.



As can be seen from the contour plot, for $f_1(x)$ the Gradient Descent algorithm converges smoothly but the other two algorithms are very tortuous. For $f_2(x)$, the New Global Search algorithm converges quickly to a minimum, and the Gradient Descent algorithm converges in a jagged fashion.

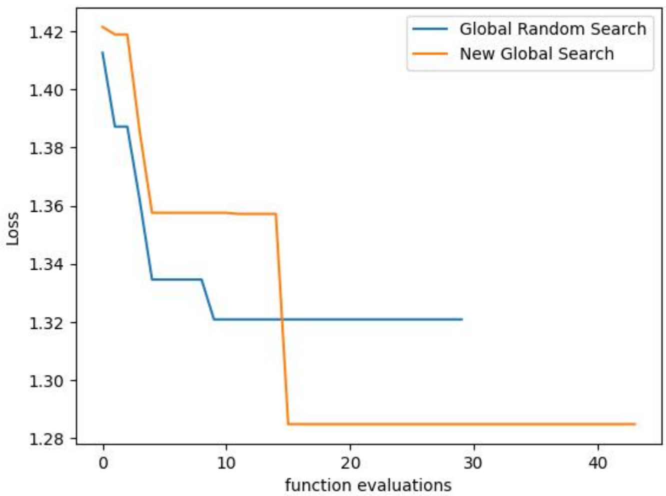




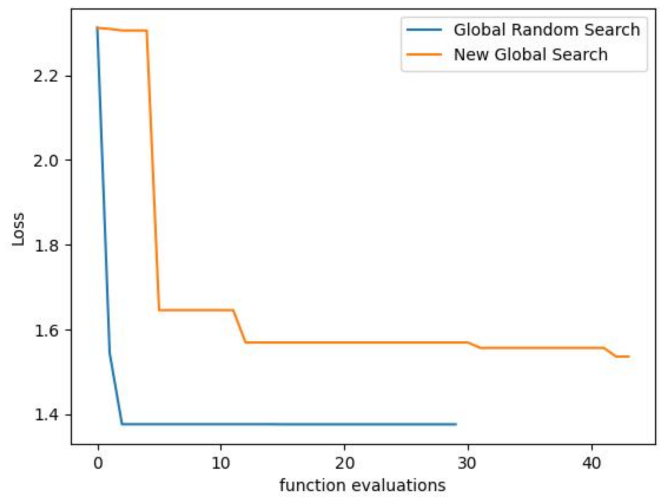
(c)

In this question, I use the downloaded conv net model and apply the Global Random Search Algorithm and the New Global Search Algorithm from the last two problems, where the parameters are chosen to be 20 for N, 5 for M, and 3 for the number of iterations, and use mini-batch size, adam parameters, and number of epochs as hyperparameters respectively.

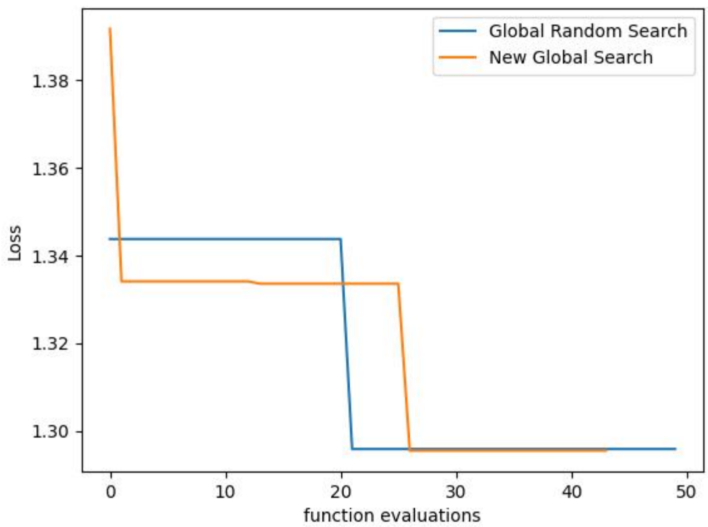
(i) The mini-batch size was chosen to range from 0 to 128, and the parameters of the fixed adam algorithm were $\alpha = 0.01$, $\beta 1 = 0.9$, $\beta 2 = 0.99$, and fixed epochs of 10. The New Global Search algorithm performs better than the Global Random Search algorithm and the best batch size is around 15.



(ii) For the Adam algorithm, I chose parameters α ranging from 0.1 to 0.001, $\beta 1$ ranging from 0.25 to 0.99, $\beta 2$ ranging from 0.9 to 0.999, a fixed small batch size of 15 and an epoch of 15 for the experiment. The global random search algorithm performed better than the new global search algorithm The global random search algorithm performed better than the new global search algorithm. The best choice of parameters was α = 0.001. β1 = 0.9 and β2 = 0.999.



(iii) I chose epochs in the range of 10 to 30. According to the experimental results of the above two problems, the specified mini-batch size is 15, Adam's parameters α = 0.001, β1 = 0.9 and β2 = 0.999. The Global Random Search algorithm performed slightly better than the New Global Search algorithm, with the best epochs being around 21.



Appendix

a(i)

```python
import sys
from random import uniform


def random_search(f, n, data_range, N):
    l = []
    u = []
    for Range in data_range:
        l.append(Range[0])
    for Range in data_range:
        u.append(Range[1])
    X = 0
    F = sys.maxsize
    x_list = []
    f_list = []
    for i in range(N):
        data_x = []
        for j in range(n):
            x = uniform(l[j], u[j])
            data_x.append(x)
        data_f = f(data_x[0], data_x[1])
        if data_f < F:
            X = data_x
            F = data_f
        x_list.append(X)
        f_list.append(F)
    return x_list, f_list
```

a(ii)

```python
from random import uniform
from timeit import timeit
import matplotlib
from a_i import random_search

matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
import sys
```

```python
def gradient_descent(f, df, n, x0, iter_times, alpha=0.01):
    x = x0
    f = f(x0[0], x0[1])
    x_list = []
    f_list = []
    x_list.append(x0)
    f_list.append(f)
    for i in range(iter_times):
        for j in range(n):
            x[j] -= alpha * df[j](x[j])
        f = f(x[0], x[1])
        x_list.append(x)
        f_list.append(f)
    return x_list, f_list


def func_1():
    f = lambda x, y: 3 * (x - 3) ** 4 + 9 * (y - 8) ** 2
    dx = lambda x: 12 * (x - 3) ** 3
    dy = lambda y: 18 * y - 144
    return f, (dx, dy)


def func_2():
    f = lambda x, y: 9 * abs(y - 8) + max(0, x - 3)
    dx = lambda x: np.heaviside(x - 3, 0)
    dy = lambda y: 9 * np.sign(y - 8)
    return f, (dx, dy)


if __name__ == '__main__':
    f1, df1 = func_1()
    f2, df2 = func_2()
    n = 2
    data_range = [[4, 8], [0, 3]]
    x0 = [7, 7]
    iter_times = 200
    New_Global_Search_N, New_Global_Search_M, New_Global_Search_n = 20, 5, 10
```

```python
    global_random_search_time1 = timeit(lambda: random_search(f1, n, data_range, N=iter_times), number=500)
    gradient_descent_time1 = timeit(lambda: gradient_descent(f1, df1, n, x0, iter_times=iter_times), number=500)
    global_random_search_time2 = timeit(lambda: random_search(f2, n, data_range, N=iter_times), number=500)
    gradient_descent_time2 = timeit(lambda: gradient_descent(f2, df2, n, x0, iter_times=iter_times), number=500)
    print(f'Global Random Search for f1(x) = {global_random_search_time1}')
    print(f'Gradient Descent for f1(x) = {gradient_descent_time1}')
    print(f'Global Random Search for f2(x) = {global_random_search_time2}')
    print(f'Gradient Descent for f2(x) = {gradient_descent_time2}')

    global_random_search_x_list, global_random_search_f_list = random_search(f1, n, data_range, N=iter_times)
    gradient_descent_x_list, gradient_descent_f_list = gradient_descent(f1, df1, n, x0, iter_times=iter_times)
    global_random_search_x_list_2, global_random_search_f_list_2 = random_search(f2, n, data_range, N=iter_times)
    gradient_descent_x_list_2, gradient_descent_f_list_2 = gradient_descent(f2, df2, n, x0, iter_times=iter_times)
    global_random_search_x_list_ = list(range(len(global_random_search_f_list)))
    gradient_descent_x_list_ = list(range(len(gradient_descent_f_list)))
    global_random_search_x_list_2_ = list(range(len(global_random_search_f_list_2)))
    gradient_descent_x_list_2_ = list(range(len(gradient_descent_f_list_2)))
    plt.plot(global_random_search_x_list_, global_random_search_f_list, label='Global Random Search Algorithm', color='tab:blue')
```

```python
    plt.plot(gradient_descent_x_list_, gradient_descent_f_list, label=f'Gradient Descent Algorithm', color='tab:red')
    plt.xlabel('function evaluations')
    plt.ylabel('f1')
    plt.legend()
    plt.show()

    plt.plot(global_random_search_x_list_2_, global_random_search_f_list_2, label='Global Random Search Algorithm',
             color='tab:blue')
    plt.plot(gradient_descent_x_list_2_, gradient_descent_f_list_2, label=f'Gradient Descent Algorithm', color='tab:red')
    plt.xlabel('function evaluations')
    plt.ylabel('f2')
    plt.legend()
    plt.show()
```

b(i)

```python
from random import uniform
import sys


def Sort(a, b):
    return map(list, zip(*sorted(zip(a, b))))


def new_global_search(f, n, data_range, N, M, itr_times):
    l = []
    u = []
    for Range in data_range:
        l.append(Range[0])
    for Range in data_range:
        u.append(Range[1])
    X = 0
    F = sys.maxsize
    x_list = []
    f_list = []
    current_x_list = []
    for i in range(N):
        tmp = []
```

```
        for j in range(n):
            tmp.append(uniform(l[j], u[j]))
        current_x_list.append(tmp)
    current_f_list = [0] * N
    for i in range(N):
        current_x = current_x_list[i]
        current_f = f(current_x[0], current_x[1])
        current_f_list[i] = current_f
        if current_f < F:
            X = current_x
            F = current_f
        x_list.append(X)
        f_list.append(F)
    opt_times = (N - M) // M
    for _ in range(itr_times):
        current_f_list, current_x_list = Sort(current_f_list, current_x_list)
        for i in range(M):
            current_x = current_x_list[i]
            for j in range(opt_times):
                x_plus = [x * uniform(0.7, 1.3) for x in current_x]
                k = M + (i * opt_times) + j
                current_x_list[k] = x_plus
                current_f_list[k] = f(x_plus[0], x_plus[1])
                if current_f_list[k] < F:
                    X = current_x_list[k]
                    F = current_f_list[k]
                x_list.append(X)
                f_list.append(F)
    return x_list, f_list
```

b(ii)

```
from timeit import timeit
import matplotlib

from a_i import random_search
from a_ii import gradient_descent
from b_i import new_global_search

matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
```

```python


def func_1():
    f = lambda x, y: 3 * (x - 3) ** 4 + 9 * (y - 8) ** 2
    dx = lambda x: 12 * (x - 3) ** 3
    dy = lambda y: 18 * y - 144
    return f, (dx, dy)


def func_2():
    f = lambda x, y: 9 * abs(y - 8) + max(0, x - 3)
    dx = lambda x: np.heaviside(x - 3, 0)
    dy = lambda y: 9 * np.sign(y - 8)
    return f, (dx, dy)


if __name__ == '__main__':
    f1, df1 = func_1()
    f2, df2 = func_2()
    n = 2
    data_range = [[4, 8], [0, 3]]
    x0 = [7, 7]
    iter_times = 200
    New_Global_Search_N, New_Global_Search_M, New_Global_Search_n = 20, 5, 10
    global_random_search_time1 = timeit(lambda: random_search(f1, n, data_range, N=iter_times), number=500)
    gradient_descent_time1 = timeit(lambda: gradient_descent(f1, df1, n, x0, iter_times=iter_times), number=500)
    new_global_search_time1 = timeit(lambda: new_global_search(f1, n, data_range, N=New_Global_Search_N, M=New_Global_Search_M,
    itr_times=New_Global_Search_n), number=500)
    global_random_search_time2 = timeit(lambda: random_search(f2, n, data_range, N=iter_times), number=500)
    gradient_descent_time2 = timeit(lambda: gradient_descent(f2, df2, n, x0, iter_times=iter_times), number=500)
```

```
    new_global_search_time2 = timeit(
        lambda: new_global_search(f2, n, data_range, N=New_Global_Search_N, M=New_Global_Search_M,
                                  itr_times=New_Global_Search_n), number=500)
    print(f'Global Random Search for f1(x) = {global_random_search_time1}')
    print(f'Gradient Descent for f1(x) = {gradient_descent_time1}')
    print(f'New Global Search for f1(x) = {new_global_search_time1}')

    print(f'Global Random Search for f2(x) = {global_random_search_time2}')
    print(f'Gradient Descent for f2(x) = {gradient_descent_time2}')
    print(f'New Global Search for f2(x) = {new_global_search_time2}')

    global_random_search_x_list, global_random_search_f_list = random_search(f1, n, data_range, N=iter_times)
    gradient_descent_x_list, gradient_descent_f_list = gradient_descent(f1, df1, n, x0, iter_times=iter_times)
    new_global_search_x_list, new_global_search_f_list = new_global_search(f1, n, data_range, New_Global_Search_N, New_Global_Search_M, New_Global_Search_n)
    global_random_search_x_list_2, global_random_search_f_list_2 = random_search(f2, n, data_range, N=iter_times)
    gradient_descent_x_list_2, gradient_descent_f_list_2 = gradient_descent(f2, df2, n, x0, iter_times=iter_times)
    new_global_search_x_list_2, new_global_search_f_list_2 = new_global_search(f1, n, data_range, New_Global_Search_N,
                                                                               New_Global_Search_M, New_Global_Search_n)

    global_random_search_x_list_ = list(range(len(global_random_search_f_list)))
```

```
    gradient_descent_x_list_ = list(range(len(gradient_descent_f_list)))
    new_global_search_x_list_ = list(range(len(new_global_search_x_list)))
    global_random_search_x_list_2_ = list(range(len(global_random_search_f_list_2)))
    gradient_descent_x_list_2_ = list(range(len(gradient_descent_f_list_2)))
    new_global_search_x_list_2_ = list(range(len(new_global_search_x_list_2)))

    plt.plot(global_random_search_x_list_, global_random_search_f_list, label='Global Random Search Algorithm',
             color='tab:blue')
    plt.plot(gradient_descent_x_list_, gradient_descent_f_list, label=f'Gradient Descent Algorithm', color='tab:red')
    plt.plot(new_global_search_x_list_, new_global_search_f_list, label=f'New Global Search Algorithm', color='tab'

        ':orange')
    plt.xlabel('function evaluations')
    plt.ylabel('f1')
    plt.legend()
    plt.show()

    plt.plot(global_random_search_x_list_2_, global_random_search_f_list_2, label='Search Random Algorithm',
             color='tab:blue')
    plt.plot(gradient_descent_x_list_2_, gradient_descent_f_list_2, label=f'Gradient Descent Algorithm',
             color='tab:red')
    plt.plot(new_global_search_x_list_, new_global_search_f_list, label=f'New Global Search Algorithm', color='tab'

        ':orange')
    plt.xlabel('function evaluations')
```

```
    plt.ylabel('f2')
    plt.legend()
    plt.show()
```

c(i)

```
from tensorflow import keras
from keras import regularizers
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D
from keras.losses import CategoricalCrossentropy
from keras.optimizers import Adam
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from b_i import new_global_search
from a_i import random_search

def get_model_loss(batch_size, alpha, beta1, beta2, epochs):
    # Model / data parameters
    num_classes = 10
    input_shape = (32, 32, 3)

    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    n = 5000
    x_train = x_train[1:n]
    y_train = y_train[1:n]
    # x_test=x_test[1:500]; y_test=y_test[1:500]

    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    print("orig x_train shape:", x_train.shape)

    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)
    model = keras.Sequential()
```

```
    model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))
    optimizer = Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
        metrics=["accuracy"])
    y_predicts = model.predict(x_test)
    loss = CategoricalCrossentropy()
    return loss(y_test, y_predicts).numpy()


if __name__ == '__main__':
    n = 5
    data_range = [
        [1, 128],
        [0.001, 0.001],
        [0.9, 0.9],
        [0.99, 0.99],
        [15, 15]
    ]
    global_random_search_x_list, global_random_search_f_list = random_search(get_model_loss, n, data_range, N=30)
    new_global_search_x_list, new_global_search_f_list = new_global_search(get_model_loss, n, data_range, N=12, M=4,
                                                                            itr_times=4)
    global_random_search_x_list_, new_global_search_x_list_ = list(range(len(global_random_search_f_list))), list(range(len(new_global_search_f_list)))
```

```python
    plt.plot(global_random_search_x_list_, global_random_search_f_list, label='Global Random Search')
    plt.plot(new_global_search_x_list_, new_global_search_f_list, label='New Global Search')
    plt.xlabel('function evaluations')
    plt.ylabel('loss')
    plt.legend()
    plt.show()
```

c(ii)

```python
from tensorflow import keras
from keras import regularizers
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D
from keras.losses import CategoricalCrossentropy
from keras.optimizers import Adam
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from b_i import new_global_search
from a_i import random_search

def get_model_loss(batch_size, alpha, beta1, beta2, epochs):
    # Model / data parameters
    num_classes = 10
    input_shape = (32, 32, 3)

    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    n = 5000
    x_train = x_train[1:n]
    y_train = y_train[1:n]
    # x_test=x_test[1:500]; y_test=y_test[1:500]

    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    print("orig x_train shape:", x_train.shape)

    # convert class vectors to binary class matrices
```

```python
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)
    model = keras.Sequential()
    model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))
    optimizer = Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
        metrics=["accuracy"])
    y_predicts = model.predict(x_test)
    loss = CategoricalCrossentropy()
    return loss(y_test, y_predicts).numpy()


if __name__ == '__main__':
    n = 5
    data_range = [
        [15, 15],
        [0.1, 0.001],
        [0.9, 0.25],
        [0.999, 0.99],
        [15, 15]
    ]
    global_random_search_x_list, global_random_search_f_list = random_search(get_model_loss, n, data_range, N=30)
    new_global_search_x_list, new_global_search_f_list = new_global_search(get_model_loss, n, data_range, N=12, M=4,
                                    itr_times=4)
```

```
    global_random_search_x_list_, new_global_search_x_list_ = list(range(len(global_random_search_f_list))), list(
        range(len(new_global_search_f_list)))
    plt.plot(global_random_search_x_list_, global_random_search_f_list, label='Global Random Search')

    plt.plot(new_global_search_x_list_, new_global_search_f_list, label='New Global Search')
    plt.xlabel('function evaluations')
    plt.ylabel('loss')
    plt.legend()
    plt.show()
```

c(iii)

```
from tensorflow import keras
from keras import regularizers
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D
from keras.losses import CategoricalCrossentropy
from keras.optimizers import Adam
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from b_i import new_global_search
from a_i import random_search

def get_model_loss(batch_size, alpha, beta1, beta2, epochs):
    # Model / data parameters
    num_classes = 10
    input_shape = (32, 32, 3)

    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    n = 5000
    x_train = x_train[1:n]
    y_train = y_train[1:n]
    # x_test=x_test[1:500]; y_test=y_test[1:500]

    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
```

```
    print("orig x_train shape:", x_train.shape)

    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)
    model = keras.Sequential()
    model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))
    optimizer = Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
        metrics=["accuracy"])
    y_predicts = model.predict(x_test)
    loss = CategoricalCrossentropy()
    return loss(y_test, y_predicts).numpy()


if __name__ == '__main__':
    n = 5
    data_range = [
        [15, 15],
        [0.001, 0.001],
        [0.9, 0.9],
        [0.999, 0.999],
        [30, 10]
    ]
    global_random_search_x_list, global_random_search_f_list = random_search(get_model_loss, n, data_range, N=30)
```

```
    new_global_search_x_list, new_global_search_f_list = new_global_search(get_model_loss, n, data_range, N=12, M=4,
                                                                            itr_times=4)
    global_random_search_x_list_, new_global_search_x_list_ = list(range(len(global_random_search_f_list))), list(
        range(len(new_global_search_f_list)))
    plt.plot(global_random_search_x_list_, global_random_search_f_list, label='Global Random Search')
    plt.plot(new_global_search_x_list_, new_global_search_f_list, label='New Global Search')
    plt.xlabel('function evaluations')
    plt.ylabel('loss')
    plt.legend()
    plt.show()
```