IE 440 - NONLINEAR MODELS IN OPERATIONS RESEARCH

Homework # 3

24.11.2023

Search Methods

TEAM OPTIMIZERS

Fatmanur Yaman - 2019402204

Ömercan Mısırlıoğlu - 2020402261

Hüseyin Emre Bacak - 2021402279

The Function

$$f(x_1, x_2) = (5x_1 - x_2)^4 + (x_1 - 2)^2 + x_1 - 2x_2 + 12$$

1. **Cyclic Coordinate Search**

- **Parameters Used**

There are 3 parameters that the Cyclic Coordinate Search method takes initially: *the function itself **f**, the initial starting point **initial_x**, and the desired accuracy error **epsilon***. In addition to these, there are two functions that help the cyclic coordinate search in finding the minimum point called golden_section and argmin functions. These functions also take some parameters.

- **Results**

```
In [262]: cyclic_search(f, np.array([0, 0]), 0.01)

x* = [ 5.81998449 29.88423598]
f(x*) = -26.977799252690232
Out[262]:
```

	k	x(k)	f(x(k))	d(k)	a(k)	x(k+1)
0	0	[0.0, 0.0]	16.000000	[1, 0]	0.103636	[0.10363558800983827, 0.0]
1	1	[0.0, 0.0]	16.000000	[0, 1]	0.793068	[0.10363558800983827, 0.7930677212935748]
2	2	[0.10363558800983827, 0.7930677212935748]	14.119408	[1, 0]	0.154327	[0.25796279597218774, 0.7930677212935748]
3	3	[0.10363558800983827, 0.7930677212935748]	14.119408	[0, 1]	0.518722	[0.25796279597218774, 1.3117896718265092]
4	4	[0.25796279597218774, 1.3117896718265092]	12.669077	[1, 0]	0.100458	[0.3584204840695533, 1.3117896718265092]
...	...	...	...	...	...	...
667	667	[5.812936863272546, 29.846970234382987]	-26.968007	[0, 1]	0.010000	[5.817909928731961, 29.85697034316769]
668	668	[5.817909928731961, 29.85697034316769]	-26.972751	[1, 0]	0.002075	[5.819984485821251, 29.85697034316769]
669	669	[5.817909928731961, 29.85697034316769]	-26.972751	[0, 1]	0.027266	[5.819984485821251, 29.884235979680284]
670	670	[5.819984485821251, 29.884235979680284]	-26.977799	[1, 0]	0.004973	[5.824957551280665, 29.884235979680284]

Case 1. (f = f(x), initial_x = np.array([0, 0]), ε = 0.01)

```
In [264]: cyclic_search(f, np.array([1000, 1000]), 0.02)

x* = [ 6.47750399 33.65623027]
f(x*) = -26.196019079845954
Out[264]:
```

	k	x(k)	f(x(k))	d(k)	a(k)	x(k+1)
0	0	[1000.0, 1000.0]	2.560000e+14	[1, 0]	-99.998807	[900.0011933801937, 1000.0]
1	1	[1000.0, 1000.0]	2.560000e+14	[0, 1]	99.998069	[900.0011933801937, 1099.9980689640877]
2	2	[900.0011933801937, 1099.9980689640877]	1.336348e+14	[1, 0]	-99.998807	[800.0023867603875, 1099.9980689640877]
3	3	[900.0011933801937, 1099.9980689640877]	1.336348e+14	[0, 1]	99.998069	[800.0023867603875, 1199.9961379281754]
4	4	[800.0023867603875, 1199.9961379281754]	6.146699e+13	[1, 0]	-99.998807	[700.0035801405812, 1199.9961379281754]
...	...	...	...	...	...	...
9995	9995	[6.477215462945153, 33.65477593445833]	-2.619608e+01	[0, 1]	-0.475573	[6.572574158204468, 33.17920285716218]
9996	9996	[6.572574158204468, 33.17920285716218]	-2.686738e+01	[1, 0]	-0.095070	[6.47750398956216, 33.17920285716218]
9997	9997	[6.572574158204468, 33.17920285716218]	-2.686738e+01	[0, 1]	0.477027	[6.47750398956216, 33.656230272469855]
9998	9998	[6.47750398956216, 33.656230272469855]	-2.619602e+01	[1, 0]	0.095359	[6.572862684821475, 33.656230272469855]

Case 2. (f = f(x), initial_x = np.array([1000,1000]), ε = 0.02)

- **The Source Code**

The code below implements the cyclic coordinate search algorithm. It starts from an initial point initial_x and iteratively updates the point to minimize the function f. Each iteration cycles through the coordinate directions and uses the argmin function to find the optimal step size along each direction. The argmin function is used to find the value of alpha that minimizes the function f(xj + alpha*e), where xj is a point and e is a direction vector. It uses the golden_section function with a large interval and a small tolerance to find the optimal alpha. This is a crucial part of the cyclic coordinate search, as it determines the optimal step size along a given direction.

The process continues until the change in the point's location is less than a specified tolerance epsilon, or a maximum number of iterations max_iter is reached. It keeps track of the iteration data and prints the final minimum point x* and its function value f(x*). The function returns a DataFrame containing detailed information about each iteration, including the current point, function value, direction vector, step size, and the updated point.

In [261]:

```python
def cyclic_search(f, initial_x, epsilon, max_iter=10000):
    x_new = np.array(initial_x, dtype=float)
    x_old = np.array([float('inf'), float('inf')], dtype=float) # This assignment is done for
                                                                # initialization of the while loop.
    e = np.array([[1, 0], [0, 1]]) # matrix of 2-D direction vectors
    k = 0 # iteration count
    data = [] # This will be filled with necessary informations
              # for keeping track of iterations.

    while np.linalg.norm(x_old - x_new) > epsilon and k < max_iter: # checking distance between x(k) and x(k+1)
        x_old = x_new.copy()

        for j in range(2): # since we have 2 dimensions in x, our range is 2.
            a = argmin(f, x_old, e[:, j])
            x_temp = x_new.copy()
            x_temp[j] = x_old[j] + a * e[j, j] # actually, this is x(k+1).

            # appending necessary informations for the desired output format
            data.append({
                'k': k,
                'x(k)': x_old.copy(),
                'f(x(k))': f(x_old),
                'd(k)': e[:, j].copy(),
                'a(k)': a,
                'x(k+1)': x_temp
            })

            x_new = x_temp.copy() # updating x's properly at the end of each iteration

        k += 1

    df = pd.DataFrame(data, columns=['k', 'x(k)', 'f(x(k))', 'd(k)', 'a(k)', 'x(k+1)'])
    print('x* = ' + str(df.iloc[-1]['x(k)']))
    print('f(x*) = ' + str(df.iloc[-1]['f(x(k))']))
    return df
```

In [259]:

```python
# we defined golden section search method to use in exact line search for
# one-dimensional optimization where we need to determine the step length
# in each iteration k.
def golden_section(a, b, epsilon, f):
    x = b - 0.618*(b-a)
    y = a + 0.618*(b-a)
    fx = f(x)
    fy = f(y)
    while b-a >= epsilon:
        if fx > fy:
            a = x
            x = y
            y = a + 0.618*(b-a)
            fx = fy
            fy = f(y)
        else:
            b = y
            y = x
            x = b - 0.618*(b-a)
            fy = fx
            fx = f(x)
    return x
```

In [260]:

```python
# this function helps us to find the alpha value which minimizes f(x(k) + alpha*d(k))
# where x(k) is the x value at the kth iteration and d(k) is the direction vector at the kth iteration.
# (for cyclic coordinate search, d(k) is either [1, 0] or [0, 1])
# (for hook & jeeves, d(k) is either [1, 0] or [0, 1]) --> exploratory moves
# (for hook & jeeves, d(k) is x_temp - x(k)) --> pattern moves
def argmin(f, xj, e):
    def g(alpha):
        return f(xj + (alpha*e))
    arg_min = golden_section(-100, 100, 0.005, g)
    return arg_min
```

2. Hook & Jeeves Method

- **Parameters Used**

The parameters used in Hook & Jeeves Method are the same as described for Cyclic Coordinate Search method.

- **Results**

In [266]:
```
hook_jeeves(f, np.array([0, 0]), 0.01)
```

```
x* = [ 6.44182768 32.99974853]
f(x*) = -27.43713076505371
```

Out[266]:

	k	x(k)	f(x(k))	x_temp	d(k)	a(k)	x(k+1)
0	0	[0.0, 0.0]	16.000000	[0.0, 0.7930677212935748]	[0.0, 0.7930677212935748]	0.000568	[0.0, 0.7935185674139287]
1	1	[0.0, 0.7935185674139287]	14.809449	[0.25739899773976543, 0.7935185674139287]	[0.25739899773976543, 0.0]	0.003810	[0.258379752374863, 0.7935185674139287]
2	2	[0.258379752374863, 0.7935185674139287]	13.766278	[0.258379752374863, 2.0856682378646525]	[0.0, 1.2921496704507238]	-0.001141	[0.258379752374863, 2.0856682378646525]
3	3	[0.258379752374863, 2.0856682378646525]	11.517273	[0.5098357800627631, 2.0856682378646525]	[0.2514560276879001, 0.0]	-0.001930	[0.5098357800627631, 2.0856682378646525]
4	4	[0.5098357800627631, 2.0856682378646525]	10.605246	[0.5098357800627631, 3.3429788145498733]	[0.0, 1.2573105766852208]	-0.001141	[0.5098357800627631, 3.3429788145498733]
...	...	...	...	...	...	...	...
788	788	[6.4405849186722515, 32.99346869748947]	-27.436983	[6.4405849186722515, 32.99533126402145]	[0.0, 0.0018625665319831342]	0.694898	[6.4405849186722515, 32.996625557543894]
789	789	[6.4405849186722515, 32.996625557543894]	-27.437021	[6.441619626923207, 32.996625557543894]	[0.0010347082509554184, 0.0]	-0.396560	[6.441209302906116, 32.996625557543894]
790	790	[6.441209302906116, 32.996625557543894]	-27.437058	[6.441209302906116, 32.99848812407588]	[0.0, 0.0018625665319831342]	0.676706	[6.441209302906116, 32.999748533678755]
791	791	[6.441209302906116, 32.999748533678755]	-27.437094	[6.442244011157071, 32.999748533678755]	[0.0010347082509554184, 0.0]	-0.402370	[6.44182767533086, 32.999748533678755]
792	792	[6.44182767533086, 32.999748533678755]	-27.437131	[6.44182767533086, 33.00161110021074]	[0.0, 0.0018625665319831342]	0.659894	[6.44182767533086, 33.0028401958213]

Case 1. (f = f(x), initial_x = np.array([0, 0]), ε = 0.01 , max_iter = 10000)

In [267]:
```
hook_jeeves(f, np.array([100, 100]), 0.001)
```

```
x* = [ 6.60276232 33.81291097]
f(x*) = -27.42988002356907
```

Out[267]:

	k	x(k)	f(x(k))	x_temp	d(k)	a(k)	x(k+1)
0	0	[100.0, 100.0]	2.560001e+10	[100.0, 199.99806896408774]	[0.0, 99.99806896408774]	3.007562	[100.0, 500.7484168257675]
1	1	[100.0, 500.7484168257675]	8.714817e+03	[100.0, 500.7937307205647]	[0.0, 0.045313894797232024]	-0.000661	[100.0, 500.79370078765504]
2	2	[100.0, 500.79370078765504]	8.714809e+03	[99.7307742798274, 500.79370078765504]	[-0.2692257201725994, 0.0]	0.000568	[99.73062122937763, 500.79370078765504]
3	3	[99.73062122937763, 500.79370078765504]	8.682414e+03	[99.73062122937763, 499.44569567729957]	[0.0, -1.3480051103554729]	-0.001696	[99.73062122937763, 499.44569567729957]
4	4	[99.73062122937763, 499.44569567729957]	8.664508e+03	[99.73062122937763, 499.44626416105984]	[0.0, 0.0005684837602757398]	0.954250	[99.73062122937763, 499.4468066365252]
...	...	...	...	...	...	...	...
2106	2106	[6.6049338031047204, 33.82387271246419]	-2.742942e+01	[6.6049338031047204, 33.818151869594246]	[0.0, -0.005720842869941123]	-0.038294	[6.6049338031047204, 33.81837094425848]
2107	2107	[6.6049338031047204, 33.81837094425848]	-2.742954e+01	[6.602784747580116, 33.81837094425848]	[-0.002149055524604683, 0.0]	-0.492016	[6.6038421163434675, 33.81837094425848]
2108	2108	[6.6038421163434675, 33.81837094425848]	-2.742965e+01	[6.6038421163434675, 33.81265010138854]	[0.0, -0.005720842869941123]	-0.045599	[6.6038421163434675, 33.81291096610701]
2109	2109	[6.6038421163434675, 33.81291096610701]	-2.742977e+01	[6.601693060818863, 33.81291096610701]	[-0.002149055524604683, 0.0]	-0.497549	[6.60276232111424, 33.81291096610701]
2110	2110	[6.602762321114244, 33.81291096610701]	-2.742988e+01	[6.602762321114244, 33.80719012323707]	[0.0, -0.005720842869941123]	-0.057408	[6.602762321114244, 33.807518542697146]

Case 2. (f = f(x), initial_x = np.array([100, 100]), ε = 0.001, max_iter = 10000)

- **The Source Code**

The provided code implements the Hook & Jeeves method, a pattern search algorithm used for finding local minima of functions in multidimensional spaces, beneficial when derivative information is not available. The algorithm starts with an initial guess initial_x and sets x_current as the current point. A matrix of 2D direction vectors [1, 0] and [0, 1] is defined for exploratory moves in each coordinate direction. For each coordinate direction, the algorithm performs an exploratory move. Using the argmin function (which employs the golden section search), it finds the step size that minimally impacts the function f in the current direction. If the function value at the new point x_new is lower than that at x_current, x_temp and x_best are updated, and improvement is set to True. If there was an improvement in the exploratory phase, a pattern move is attempted. The pattern direction is the difference between x_best and x_current. The loop breaks if the norm of the difference between x_best and x_current is less than epsilon or if the maximum iteration count max_iter is exceeded.

In [265]:
```
def hook_jeeves(f, initial_x, epsilon, max_iter=10000):
    x_current = np.array(initial_x, dtype=float)
    e = np.array([[1, 0], [0, 1]]) # matrix of 2-D direction vectors
    k = 0 # iteration count
    data = [] # This will be filled with necessary informations
              # for keeping track of iterations.

    while True:
        x_best = np.copy(x_current)
        x_temp = None
        improvement = False

        # Exploratory moves
        for j in range(len(x_current)):
            direction = e[:, j] # jth column of direction vector. (for this
                                # problem, it can only be [1, 0] or [0, 1])
            a = argmin(f, x_current, direction)
            x_new = np.copy(x_current)
            x_new[j] += a # temporary x after exploratory moves

            if f(x_new) < f(x_current): # if we see that there is improvement in exploratory moves,
                                        # we adjust x_temp and x_best.
                x_temp = np.copy(x_new)
                x_best = x_new
                improvement = True

        if improvement:
            # Pattern move
            pattern_direction = x_best - x_current
            a = argmin(f, x_best, pattern_direction)
            x_pattern = x_best + a * pattern_direction

            if f(x_pattern) < f(x_best):
                x_best = x_pattern
```

```
            if f(x_pattern) < f(x_best):
                x_best = x_pattern

            # appending necessary informations for the desired output format
            data.append({
                'k': k,
                'x(k)': np.copy(x_current),
                'f(x(k))': f(x_current),
                'x_temp': np.copy(x_temp),
                'd(k)': np.copy(pattern_direction),
                'a(k)': a,
                'x(k+1)': np.copy(x_best)
            })

            x_current = x_best # updating current x accordingly
        else:

            if np.linalg.norm(x_best - x_current) < epsilon or k > max_iter: # checking distance between x(k) and x(k+1)
                break

        k += 1

    df = pd.DataFrame(data, columns=['k', 'x(k)', 'f(x(k))', 'x_temp', 'd(k)', 'a(k)', 'x(k+1)'])
    print('x* = ' + str(df.iloc[-1]['x(k)']))
    print('f(x*) = ' + str(df.iloc[-1]['f(x(k))']))
    return df
```

3. Simplex Search Method

- **Parameters Used**

The simplex search method takes 5 parameters. The parameter f represents the function we try to minimize. Parameters α, β, and γ represent reflection, contraction, and expansion coefficients respectively. Since we want to execute the simplex search, we need 3 (which comes from n+1) points, provided with a parameter called initial_simplex.

- **Results**

In [273]:
```
initial_simplex = [np.array([0, 0]), np.array([1, 0]), np.array([0, 1])]
simplex_search(f, initial_simplex, 1, 2, 0.5, 0.1)
```

```
x* = [ 6.66772842 34.04497337]
f(x*) = -27.385624641255227
```

Out[273]:

	Iteration	x_bar	x_h	x_l	x_new	f(x_new)	type
0	0	[0.0, 0.5]	[1.0, 0.0]	[0.0, 1.0]	[0.5, 0.25]	39.878906	C
1	1	[0.0, 0.5]	[0.5, 0.25]	[0.0, 1.0]	[0.25, 0.375]	15.148682	C
2	2	[0.125, 0.6875]	[0.0, 0.0]	[0.0, 1.0]	[0.375, 2.0625]	10.891861	E
3	3	[0.1875, 1.53125]	[0.25, 0.375]	[0.375, 2.0625]	[0.21875, 0.953125]	13.485743	C
4	4	[0.296875, 1.5078125]	[0.0, 1.0]	[0.375, 2.0625]	[0.59375, 2.015625]	11.365315	R
5	5	[0.484375, 2.0390625]	[0.21875, 0.953125]	[0.375, 2.0625]	[1.015625, 4.2109375]	6.128269	E
6	6	[0.6953125, 3.13671875]	[0.59375, 2.015625]	[1.015625, 4.2109375]	[0.8984375, 5.37890625]	3.972286	E
7	7	[0.95703125, 4.794921875]	[0.375, 2.0625]	[0.8984375, 5.37890625]	[2.12109375, 10.259765625]	-6.369491	E
8	8	[1.509765625, 7.8193359375]	[1.015625, 4.2109375]	[2.12109375, 10.259765625]	[2.00390625, 11.427734375]	-4.919115	R
9	9	[2.0625, 10.84375]	[0.8984375, 5.37890625]	[2.12109375, 10.259765625]	[4.390625, 21.7734375]	-21.440120	E
10	10	[3.255859375, 16.0166015625]	[2.00390625, 11.427734375]	[4.390625, 21.7734375]	[2.6298828125, 13.72216796875]	-12.310086	C
11	11	[3.51025390625, 17.747802734375]	[2.12109375, 10.259765625]	[4.390625, 21.7734375]	[6.28857421875, 32.723876953125]	-26.074509	E
12	12	[5.339599609375, 27.2486572265625]	[2.6298828125, 13.72216796875]	[6.28857421875, 32.723876953125]	[8.04931640625, 40.775146484375]	-24.828694	R
13	13	[7.1689453125, 36.74951171875]	[4.390625, 21.7734375]	[6.28857421875, 32.723876953125]	[5.77978515625, 29.261474609375]	-26.439111	C
14	14	[6.0341796875, 30.99267578125]	[8.04931640625, 40.775146484375]	[5.77978515625, 29.261474609375]	[7.041748046875, 35.8839111328125]	-27.099046	C
15	15	[6.4107666015625, 32.57269287109375]	[6.28857421875, 32.723876953125]	[7.041748046875, 35.8839111328125]	[6.34967041015625, 32.648284912109375]	-27.371362	C
16	16	[6.695709228515625, 34.26609802246094]	[5.77978515625, 29.261474609375]	[6.34967041015625, 32.648284912109375]	[6.2377471923828125, 31.76378631591797]	-27.221973	C
17	17	[6.293708801269531, 32.20603561401367]	[7.041748046875, 35.8839111328125]	[6.34967041015625, 32.648284912109375]	[6.667728424072266, 34.044973373413086]	-27.385625	C

Case 1. (f = f(x), initial_x = initial_simplex **,** alpha=1, beta=2, gamma=0.5, ε = 0.1, max_iter = 10000)

initial_simplex = [np.array([0, 0]), np.array([1, 0]), np.array([0, 1])]

```
In [278]:  initial_simplex = [np.array([1, 0]), np.array([1, 1]), np.array([2, 1])]
           simplex_search(f, initial_simplex, 1, 2, 0.5, 0.1)

           x* = [ 6.23779297 31.62768555]
           f(x*) = -27.02164192412016
Out[278]:
```

	Iteration	x_bar	x_h	x_l	x_new	f(x_new)	type
0	0	[1.0, 0.5]	[2.0, 1.0]	[1.0, 1.0]	[0.0, 0.0]	16.000000	E
1	1	[0.5, 0.5]	[1.0, 0.0]	[0.0, 0.0]	[0.0, 1.0]	15.000000	E
2	2	[0.0, 0.5]	[1.0, 1.0]	[0.0, 1.0]	[0.5, 0.75]	22.628906	C
3	3	[0.0, 0.5]	[0.5, 0.75]	[0.0, 1.0]	[0.25, 0.625]	14.215088	C
4	4	[0.125, 0.8125]	[0.0, 0.0]	[0.25, 0.625]	[0.375, 2.4375]	10.240738	E
5	5	[0.3125, 1.53125]	[0.0, 1.0]	[0.375, 2.4375]	[0.625, 2.0625]	11.665054	R
6	6	[0.5, 2.25]	[0.25, 0.625]	[0.375, 2.4375]	[1.0, 5.5]	3.062500	E
7	7	[0.6875, 3.96875]	[0.625, 2.0625]	[1.0, 5.5]	[0.65625, 3.015625]	8.435642	C
8	8	[0.828125, 4.2578125]	[0.375, 2.4375]	[1.0, 5.5]	[1.734375, 7.8984375]	-1.634093	E
9	9	[1.3671875, 6.69921875]	[0.65625, 3.015625]	[1.734375, 7.8984375]	[2.7890625, 14.06640625]	-12.720915	E
10	10	[2.26171875, 10.982421875]	[1.0, 5.5]	[2.7890625, 14.06640625]	[3.5234375, 16.46484375]	-13.322080	E
11	11	[3.15625, 15.265625]	[1.734375, 7.8984375]	[3.5234375, 16.46484375]	[6.0, 30.0]	-26.000000	E
12	12	[4.76171875, 23.232421875]	[2.7890625, 14.06640625]	[6.0, 30.0]	[6.734375, 32.3984375]	-21.018467	R
13	13	[6.3671875, 31.19921875]	[3.5234375, 16.46484375]	[6.0, 30.0]	[4.9453125, 23.83203125]	-21.403586	C
14	14	[5.47265625, 26.916015625]	[6.734375, 32.3984375]	[6.0, 30.0]	[4.2109375, 21.43359375]	-21.747393	R
15	15	[5.10546875, 25.716796875]	[4.9453125, 23.83203125]	[6.0, 30.0]	[5.265625, 27.6015625]	-24.643467	R
16	16	[5.6328125, 28.80078125]	[4.2109375, 21.43359375]	[6.0, 30.0]	[7.0546875, 36.16796875]	-27.091086	E
17	17	[6.52734375, 33.083984375]	[5.265625, 27.6015625]	[7.0546875, 36.16796875]	[5.896484375, 30.3427734375]	-27.058569	C
18	18	[6.4755859375, 33.25537109375]	[6.0, 30.0]	[7.0546875, 36.16796875]	[6.23779296875, 31.627685546875]	-27.021642	C

Case 2. (f = f(x), initial_x = initial_simplex **,** alpha=1, beta=2, gamma=0.5, ε = 0.1, max_iter = 10000)

initial_simplex = [np.array([1, 0]), np.array([1, 1]), np.array([2, 1])]

- **The Source Code**

The simplex search method is an algorithm that uses n + 1 starting points (vertices) to solve an n-dimensional optimization problem and is based on continuously changing these starting points. There are 3 types of modification methods: contraction, expansion, and reflection. In each iteration, the algorithm evaluates the objective function at these vertices and moves the simplex towards the region of better solutions by applying operations such as reflection, expansion, and contraction. These operations adjust the vertices of the simplex, gradually honing in on the optimal solution. The method is particularly effective for problems where the objective function is well-behaved, but it may struggle with functions that have many local minima. The implementation of the method is provided below.

In [272]:

```python
def simplex_search(f, initial_simplex, alpha=1, gamma=2, beta=0.5, epsilon=0.1, max_iter=10000):
    simplex = np.array(initial_simplex, dtype=float)
    k = 0 # Iteration count
    data = [] # This will be filled with necessary informations
              # for keeping track of iterations.

    while True:
        simplex = sorted(simplex, key=f)  # Order the simplex by function values
        x_l = simplex[0]
        x_h = simplex[-1]
        x_bar = np.mean(simplex[:-1], axis=0) # x-bar value excluding x_h

        # Reflection
        x_r = x_bar + alpha * (x_bar - x_h)
        operation = 'R'

        if f(x_r) < f(x_l):
            # Expansion
            x_e = x_bar + gamma * (x_r - x_bar)
            operation = 'E'
            x_new = x_e if f(x_e) < f(x_r) else x_r
        elif all(f(x_r) >= f(s) for s in simplex if not np.array_equal(s, x_h)): # Checking all
                                                                                  # points in the simplex
                                                                                  # excluding x_h
            # Contraction
            x_c = x_bar + beta * (x_h - x_bar)
            operation = 'C'
            x_new = x_c if f(x_c) < f(x_h) else x_h
        else:
            x_new = x_r
```

```python
        simplex_index = next(i for i, v in enumerate(simplex) if np.array_equal(v, x_h)) # Getting the index
                                                                                          # of x_h
        simplex[simplex_index] = x_new # Replacing the x_h with x_new

        # appending necessary informations for the desired output format
        data.append({
            'Iteration': k,
            'x_bar': x_bar,
            'x_h': x_h,
            'x_l': x_l,
            'x_new': x_new,
            'f(x_new)': f(x_new),
            'type': operation
        })

        k += 1
        f_values = np.array([f(x) for x in simplex])

        if np.std(f_values) < epsilon or k > max_iter: # Stopping condition for simplex search is
                                                       # checking the standard deviation of points in the simplex.
            break

    df = pd.DataFrame(data)
    print('x* = ' + str(df.iloc[-1]['x_new']))
    print('f(x*) = ' + str(df.iloc[-1]['f(x_new)']))
    return df
```