Homework 1

Daniil Sherki

MSc-1, Petroleum Engineering

1 Optimization problem example – 1 point

Find the dimensions (height h and radius r) that will minimize the surface area of the metal to manufacture a circular cylindrical can of volume V .

Solution

In this case we assume that *V* some constatnt and we can calculate one of parameter (*h* or *r*) from this equation

From school math course we know how calculated cylindrical colume and surface of area.

Thus, we solve some system of equtions, where one equation need to optimizing.

$$\begin{cases} V = \pi r^2 h \\ S_{\text{surf}} = 2\pi r^2 + 2\pi r h \end{cases} \tag{1}$$

We need get *h* from first equation and put it in second.

$$\begin{cases} h = \dfrac{V}{\pi r^2} \\ S_{\text{surf}} = 2\pi r^2 + 2\pi r h \end{cases} \tag{2}$$

$$S_{\text{surf}} = 2\pi r^2 + 2\pi r \frac{V}{\pi r^2} = 2\pi r^2 + \frac{2V}{r} \tag{3}$$

Obviously that optimum point can be found when objection function derivate will bw equal to zero.

$\frac{dS_{\text{surf}}}{dr} = 4\pi r - \frac{2V}{r^2} \Rightarrow \frac{dS_{\text{surf}}}{dr} = 0 : 4\pi r - \frac{2V}{r^2} = 0 \; 4\pi r = \frac{2V}{r^2} \Rightarrow r^3 = \frac{V}{2\pi}$

$$r = \sqrt[3]{\frac{V}{2\pi}} \tag{4}$$

And *h* will be equal:

$$h = \frac{V}{\pi r^2} = \frac{V}{\pi (\frac{V}{2\pi})^{\frac{2}{3}}} \tag{5}$$

Answer: $r = \sqrt[3]{\frac{V}{2\pi}}, \; h = \frac{V}{\pi (\frac{V}{2\pi})^{\frac{2}{3}}}$

2 Optimality conditions – 3 points

Consider the unconstrained optimization problem to minimize the function,

$$f(x_1, x_2) = \frac{3}{2}(x_1^2 + x_2^2) + (1 + a)x_1 x_2 - (x_1 + x_2) + b, \&a, b \in \mathbb{R} \tag{6}$$

over $\mathbb{R}^2$, where *a* and *b* are real-valued parameters. Find all values of *a* and *b* such that the problem has a unique optimal solution.

Solution

There is need to find derivations for solve this task.

$$\begin{cases} \dfrac{\partial f}{\partial x_1} = 3x_1 + (1 + a)x_2 - 1 \\ \dfrac{\partial f}{\partial x_2} = 3x_2 + (1 + a)x_1 - 1 \end{cases} \tag{7}$$

$$\begin{cases} 3x_1 + (1 + a)x_2 - 1 = 0 \\ 3x_2 + (1 + a)x_1 - 1 = 0 \end{cases} \tag{8}$$

If we solve this linear equation, we get that

$$3x_1 + (1 + a)x_2 - 1 = 3x_2 + (1 + a)x_1 - 1 \tag{9}$$

$$x_1 = x_2 \tag{10}$$

And it's mean that optimal point (min or max) achieve only when $x_1 = x_2$

Stationary point will be equal to

$$\begin{cases} 3x_1 + (1 + a)x_1 - 1 = 0 \end{cases}$$

$$\begin{cases} 3x_2 + (1+a)x_2 - 1 = 0 \end{cases} \tag{11}$$

$$\begin{cases} x_1 = \dfrac{1}{4+a} \\ x_2 = \dfrac{1}{4+a} \end{cases} \tag{12}$$

Let's calculate second order partical derivations

$$\begin{cases} \dfrac{\partial^2 f}{\partial x_1^2} = 3 \\ \dfrac{\partial^2 f}{\partial x_1 \partial x_2} = 1 + a \\ \dfrac{\partial f}{\partial x_2^2} = 3 \end{cases} \tag{13}$$

And we need find determinant second order partical derivations in stationary point

$$\begin{vmatrix} \frac{\partial^2 f}{\partial x_1^2}(x_1^{(0)}; x_2^{(0)}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x_1^{(0)}; x_2^{(0)}) \\ \frac{\partial^2 f}{\partial x_1 \partial x_2}(x_1^{(0)}; x_2^{(0)}) & \frac{\partial f}{\partial x_2^2}(x_1^{(0)}; x_2^{(0)}) \end{vmatrix} \tag{14}$$

$$\begin{vmatrix} 3 & 1+a \\ 1+a & 3 \end{vmatrix} = 9 - (1+a)^2 \tag{15}$$

And there is three different cases:

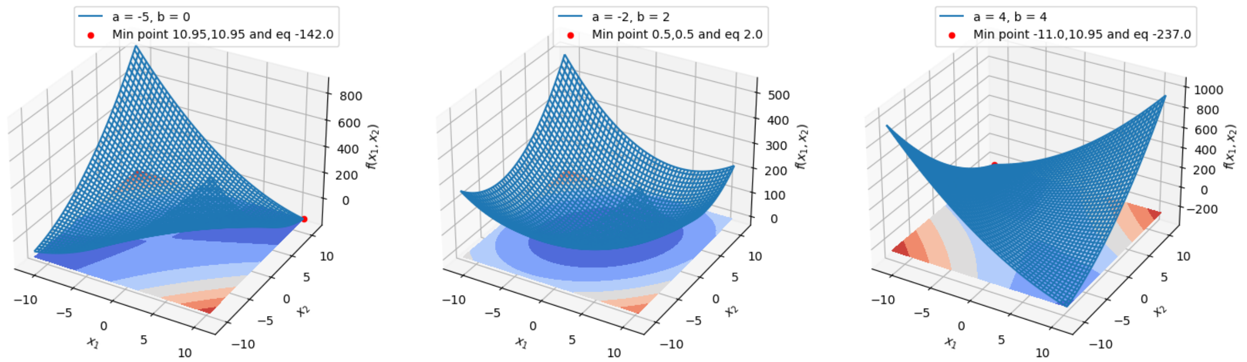
- if $9 - (1+a)^2 > 0$ then there is unique extremum in $\left(\frac{1}{4+a}; \frac{1}{4+a}\right)$ point
- if $9 - (1+a)^2 < 0$ then there is no extremum in $\left(\frac{1}{4+a}; \frac{1}{4+a}\right)$ point
- if $9 - (1+a)^2 = 0$ then we need to do more investigation

$9 - (1+a)^2 > 0 \rightarrow (1+a)^2 < 9 \rightarrow (a-2)(a+4) < 0 \rightarrow a \in (-4; 2)$

Answer: $a \in (-4; 2)$ and $b \in \mathbb{R}$

In [1]:

```python
import matplotlib.pyplot as plt
import numpy as np

def given_func(x,y,a,b):
    out = 1.5*(np.power(x,2)+np.power(y,2))+(1+a)*x*y-(x+y)+b
    return out


a = [-5, -2, 4]
b = [0, 2, 4]
xmin, xmax, step = -11, 11, 0.05
x = np.arange(xmin, xmax, step )
y = np.arange(xmin, xmax, step )
xgrid, ygrid = np.meshgrid(x, y)
plt.figure(figsize = (len(a)*6,6*2))

def min_vals_searching(xgrid, ygrid, zgrid):
    min_z = np.min(zgrid)
    for i in range(zgrid.shape[0]):
        for j in range(zgrid.shape[1]):
            if zgrid[i,j] == min_z:
                min_x, min_y = xgrid[i,j], ygrid[i,j]
                break
    return min_x, min_y, min_z

for i in range(len(a)):
    ax_3d = plt.subplot(1,len(a),i+1, projection='3d')
    zgrid = given_func(xgrid,ygrid, a[i], b[i])
    x_m, y_m, z_m = min_vals_searching(xgrid, ygrid, zgrid)
    ax_3d.plot_wireframe(xgrid, ygrid, zgrid, label = f'a = {a[i]}, b = {b[i]}')
    ax_3d.contourf(xgrid, ygrid, zgrid, zdir='z', offset=z_m, cmap='coolwarm')
    ax_3d.scatter3D(x_m, y_m, z_m,
                    label = f'Min point {np.round(x_m,2)},\
{np.round(y_m,2)} and eq {np.round(z_m)}', c='r')

    ax_3d.set_xlabel('$x_1$')
    ax_3d.set_ylabel('$x_2$')
    ax_3d.set_zlabel('$f(x_1,x_2)$')
    ax_3d.legend();

```



3 Nelder–Mead method – 8 points

Implement Nelder–Mead method for the Mishra's Bird function

$$f(x, y) = \sin(y)e^{(1-\cos(x))^2} + \cos(x)e^{(1-\sin(y))^2} + (x - y)^2 \tag{16}$$

subjected to,

$$(x + 5)^2 + (y + 5)^2 < 25 \tag{17}$$

1. To illustrate the behavior of the method, plot simplex (triangle) for every iteration.
2. Demonstrate that the algorithm may converge to different points depending on the starting point. Report explicitly two distinct starting points x^0 and the corresponding x^*.
3. Examine the behavior of the method for various parameters α, β, and γ. For one chosen x^0 show that the method may converge to different points. Report parameter values and x^*.

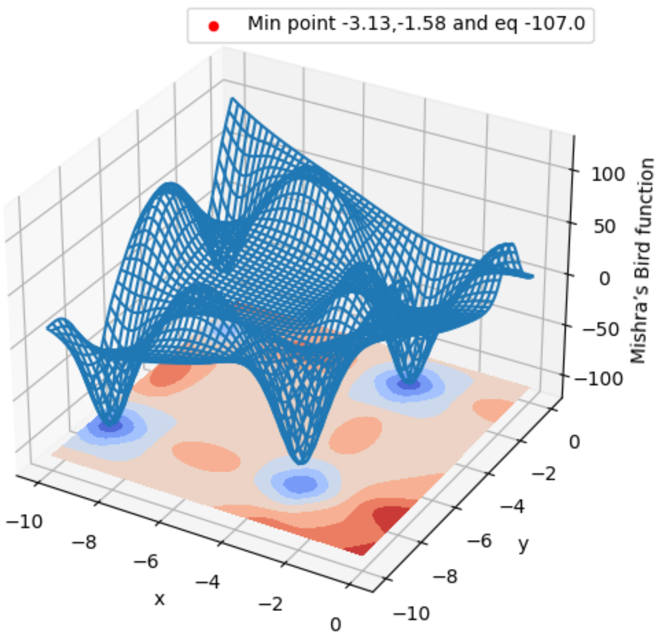
Solution

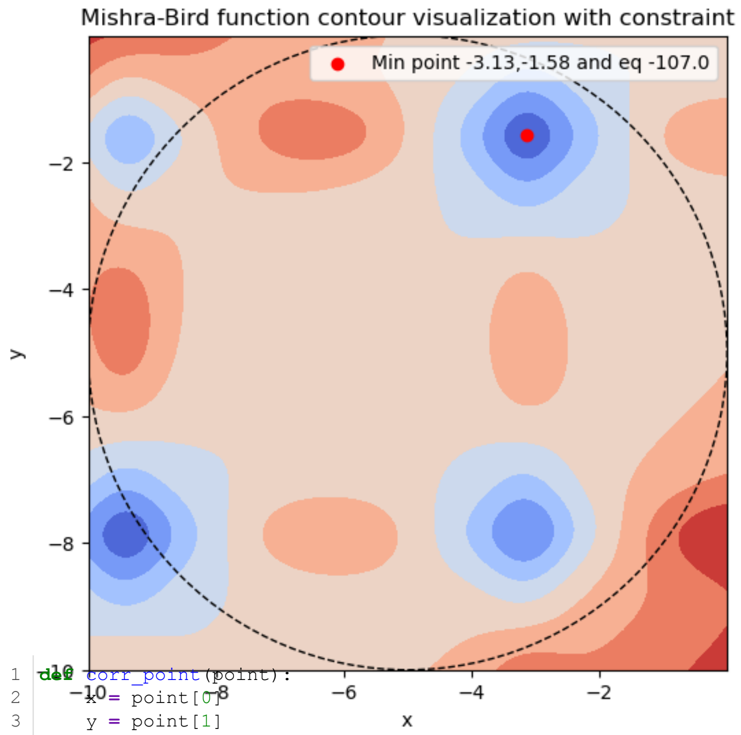
Problem statement

Mishra's bird function is the convential test function for oprimizitaion algorithm problems

In [2]:

```python
import matplotlib


def task3_objective(x,y):
    out = np.sin(y)*np.exp(np.power(1 - np.cos(x),2)) + np.cos(x)*np.exp(np.power(1 - np.sin(y),2)) + np.power(x-y,2)
    return out


xmin, xmax, step = -10, 0, 0.01
x = np.arange(xmin, xmax, step )
y = np.arange(xmin, xmax, step )
xgrid, ygrid = np.meshgrid(x, y)
zgrid = task3_objective(xgrid, ygrid)

fig = plt.figure(figsize=(6,6))
ax3d = fig.add_subplot(projection = '3d')
ax3d.plot_wireframe(xgrid, ygrid, zgrid)
x_m, y_m, z_m = min_vals_searching(xgrid, ygrid, zgrid)
ax3d.scatter3D(x_m,y_m,z_m,
               label = f'Min point {np.round(x_m,2)},\
{np.round(y_m,2)} and eq {np.round(z_m)}', c='r')

ax3d.contourf(xgrid, ygrid, zgrid, zdir='z', offset=z_m-1, cmap='coolwarm')
ax3d.set_xlabel('x')
ax3d.set_ylabel('y')
ax3d.set_zlabel('Mishra’s Bird function')
plt.legend();
plt.show();

fig = plt.figure(figsize=(6,6))
ax2d = fig.add_subplot()
ax2d.contourf(xgrid, ygrid, zgrid, cmap='coolwarm')
circle = matplotlib.patches.Circle((-5,-5), radius=5, fill = False, ls = '--')
ax2d.add_artist(circle)
ax2d.scatter(x_m,y_m,
             label = f'Min point {np.round(x_m,2)},\
{np.round(y_m,2)} and eq {np.round(z_m)}', c='r')

ax2d.set_xlabel('x')
ax2d.set_ylabel('y')
plt.legend();
plt.title('Mishra-Bird function contour visualization with constraint')
plt.show();
```





algorithm, when this point data does not fit the constraint (does not fit into

```python
def corr_point(point):
    x = point[0]
    y = point[1]
    if ((x + 5) ** 2+(y + 5) ** 2) > 25:
        vxa = x + 5
        vya = y + 5
        magv = np.sqrt(vxa ** 2 + vya ** 2)
        corr_point = np.zeros_like(point)
        corr_point[0] = -5 + (vxa / magv) * 5
        corr_point[1] = -5 + (vya / magv) * 5
    else:
        corr_point = point
    return corr_point
```

Nelder-Mead algorithm implementation (functional approach)

In [4]:

```
def nelder_mead_alg(simplex, alpha=1, gamma=2, beta=0.5, tol=1e-2):
    #grid settings for plotting
    xmin, xmax, step = -10, 0, 0.01
    x = np.arange(xmin, xmax, step )
    y = np.arange(xmin, xmax, step )
    xgrid, ygrid = np.meshgrid(x, y)
    zgrid = task3_objective(xgrid, ygrid)
    fig = plt.figure(figsize=(6,6))
    ax2d = fig.add_subplot()
    ax2d.contourf(xgrid, ygrid, zgrid, cmap='coolwarm')
    circle = matplotlib.patches.Circle((-5,-5), radius=5, fill = False, ls = '--')
    ax2d.add_artist(circle)
    ax2d.scatter(x_m,y_m,
                 label = f'Min point {np.round(x_m,2)},\
    {np.round(y_m,2)} and eq {np.round(z_m)}', c='r')

    ax2d.set_xlabel('x')
    ax2d.set_ylabel('y')
    plt.title(f'Nelder-Mead algorithm with α ={alpha}, γ ={gamma}, β = {beta}')
    plt.legend();

    oracle = 0
    k = 0
    x_h, x_l = simplex[0], simplex[1]

    while np.linalg.norm(x_h - x_l,2)>tol:
        # pre-processing
        func_val = []
        for item in simplex:
            func_val.append(task3_objective(item[0], item[1]))
            oracle += 1

        #sort
        f_h = np.max(func_val)
        f_l = np.min(func_val)
        for item in func_val:
             if item != f_h and item != f_l:
                    f_g = item
        for i in range(len(func_val)):
            if func_val[i] == f_h:
                x_h = simplex[i]
            elif func_val[i] == f_g:
                x_g = simplex[i]
            else:
                x_l = simplex[i]

        simp_plot = np.array([x_l, x_g, x_h])
        p = matplotlib.patches.Polygon(simp_plot,facecolor='None', edgecolor='black')
        ax2d.add_artist(p)
        for i in [x_l, x_g, x_h]:
            ax2d.scatter(i[0], i[1], c='black',s=5)
        #centroid
        x_c = corr_point(1./2 * (x_l + x_g))

        #reflection
        x_r = corr_point(x_c + alpha * (x_c - x_h))
        f_r = task3_objective(x_r[0], x_r[1])
        oracle += 1

        #comparsion
        if f_r < f_l:
            x_e = corr_point(x_c + gamma*(x_r - x_c))
            f_e = task3_objective(x_e[0], x_e[1])
            oracle += 1
            if f_e<f_l:
                buf = x_h
                x_h = corr_point(x_e)
                x_e = corr_point(buf)
            else:
                buf = x_h
                x_h = corr_point(x_r)
                x_r = corr_point(buf)
        elif (f_l < f_r) and (f_r < f_g):
            buf = x_h
            x_h = corr_point(x_r)
            x_r = buf
        elif (f_h > f_r) and (f_r > f_g):
            buf = x_h
            x_h = corr_point(x_r)
            x_r = corr_point(buf)
            # contraction
            x_s = x_c + beta*(x_h - x_c)
            f_s = task3_objective(x_s[0], x_s[1])
            oracle += 1
            if f_s < f_h:
                buf = x_h
                x_h = corr_point(x_s)
```

```
                x_s = corr_point(buf)
            else:
                x_g = corr_point(x_l + (x_g - x_l)/2)
                x_h = corr_point(x_l + (x_h - x_l)/2)
        else:
            # contraction
            x_s = corr_point(x_c + beta*(x_h - x_c))
            f_s = task3_objective(x_s[0], x_s[1])
            oracle += 1
            if f_s < f_h:
                buf = x_h
                x_h = corr_point(x_s)
                x_s = corr_point(buf)
            else:
                x_g = corr_point(x_l + (x_g - x_l)/2)
                x_h = corr_point(x_l + (x_h - x_l)/2)
        k += 1
        simplex = [x_l, x_g, x_h]


    x_star = np.mean([x_l, x_g, x_h],axis=0)
    f_star = task3_objective(x_star[0], x_star[1])

    print('Nelder-Mead algorithm completed')
    print(f'x* = {x_star[0]}, y* = {x_star[1]}')
    print(f'f(x*, y*) = {f_star}')
    print(f'Completed in {k} iterations')
    print(f'Oracle calls {oracle}')

    return x_star, f_star

```

Experiments

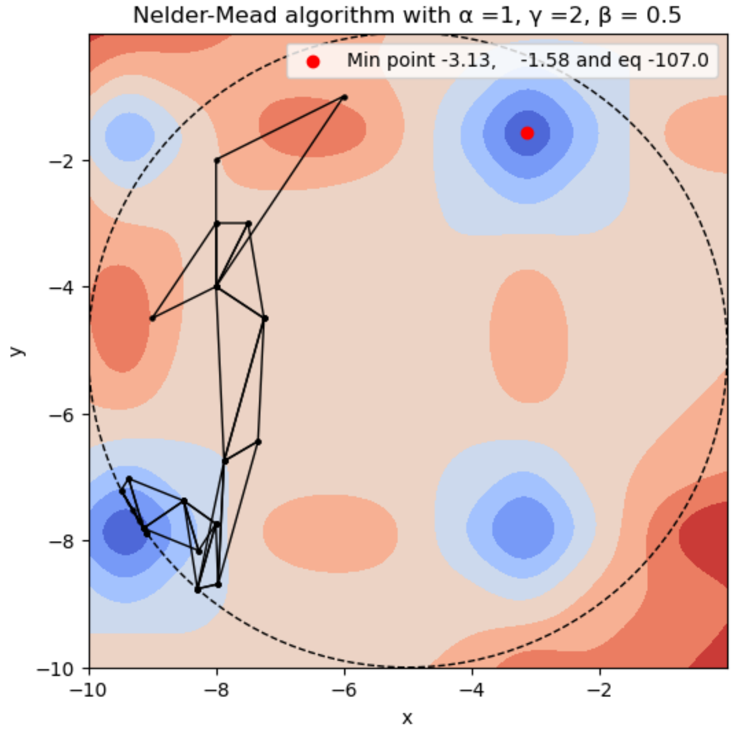
Table 1. Experiments with different initial points for Nelder-Mead method

	1 point	2 point	3 point
Case 1	(-8;-4)	(-6;-1)	(-8;-2)
Case 2	(-3;-4)	(-4;-5)	(-5;-3)
Case 3	(-4;-7)	(-6;-7)	(-5;-5)
Case 4	(-5;-5)	(-4;-3)	(-2;-1.5)

In [5]:

```
# case 1
x, f = nelder_mead_alg([np.array([-8,-4]), np.array([-6,-1]), np.array([-8,-2])])
```

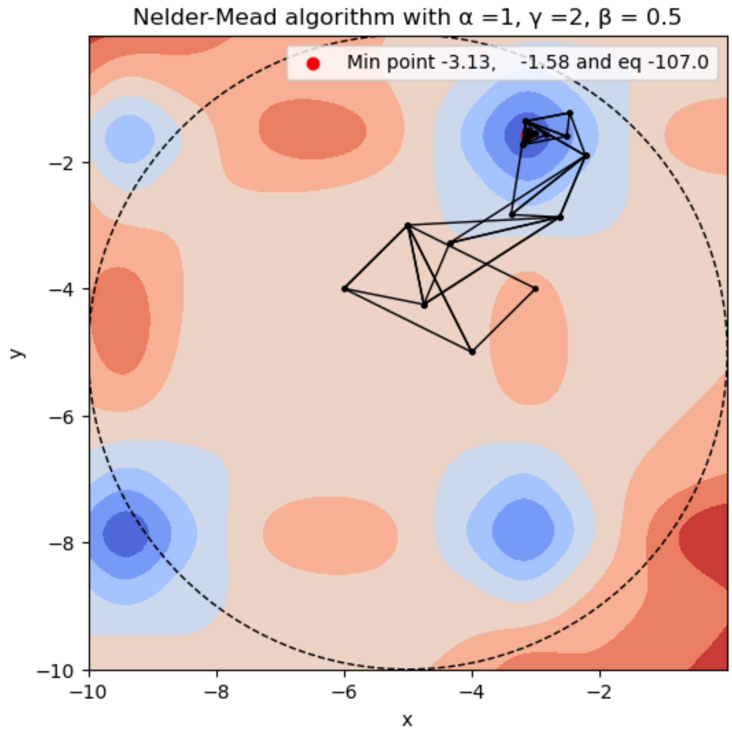
```
Nelder-Mead algorithm completed
x* = -9.184507854789997, y* = -7.725488035415727
f(x*, y*) = -97.49448316884678
Completed in 17 iterations
Oracle calls 82
```



In [6]:

```
# case 2
x, f = nelder_mead_alg([np.array([-3,-4]), np.array([-4,-5]), np.array([-5,-3])])
```

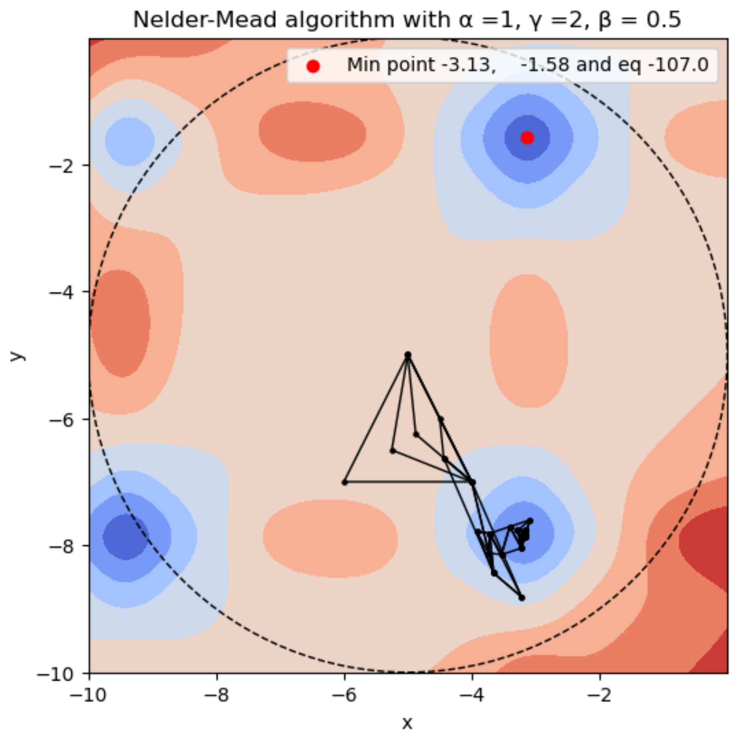
```
Nelder-Mead algorithm completed
x* = -3.1312062450387352, y* = -1.5811490956026952
f(x*, y*) = -106.76427293825802
Completed in 24 iterations
Oracle calls 119
```



In [7]:

```
# case 3
x, f = nelder_mead_alg([np.array([-4,-7]), np.array([-6,-7]), np.array([-5,-5])])
```

```
Nelder-Mead algorithm completed
x* = -3.1730365925468504, y* = -7.817926929953198
f(x*, y*) = -87.30939968840491
Completed in 24 iterations
Oracle calls 118
```



In [8]:

```
# case 4
x, f = nelder_mead_alg([np.array([-5,-5]), np.array([-4,-3]), np.array([-2,-1.5])])
```

```
Nelder-Mead algorithm completed
x* = -3.133276817961477, y* = -1.580811672342368
f(x*, y*) = -106.76302424452379
Completed in 18 iterations
Oracle calls 88
```

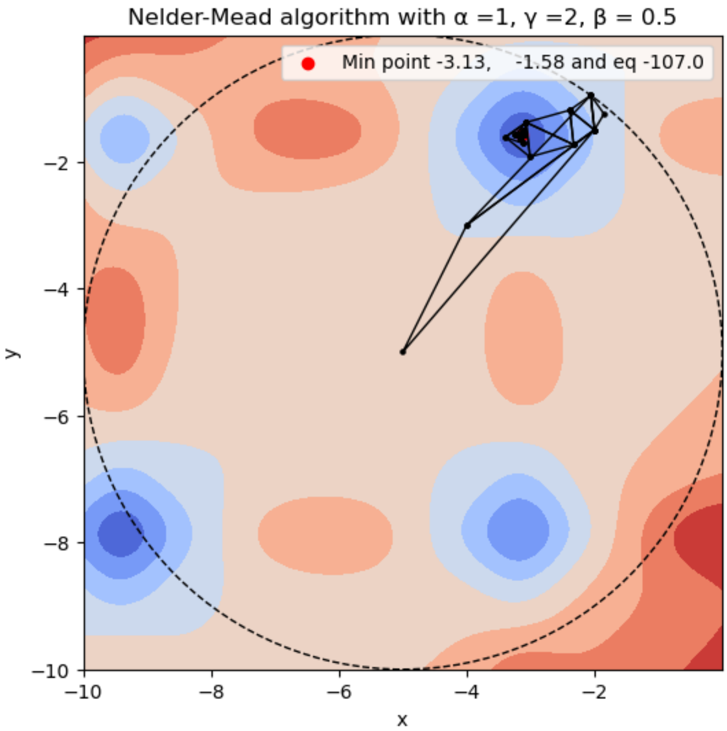


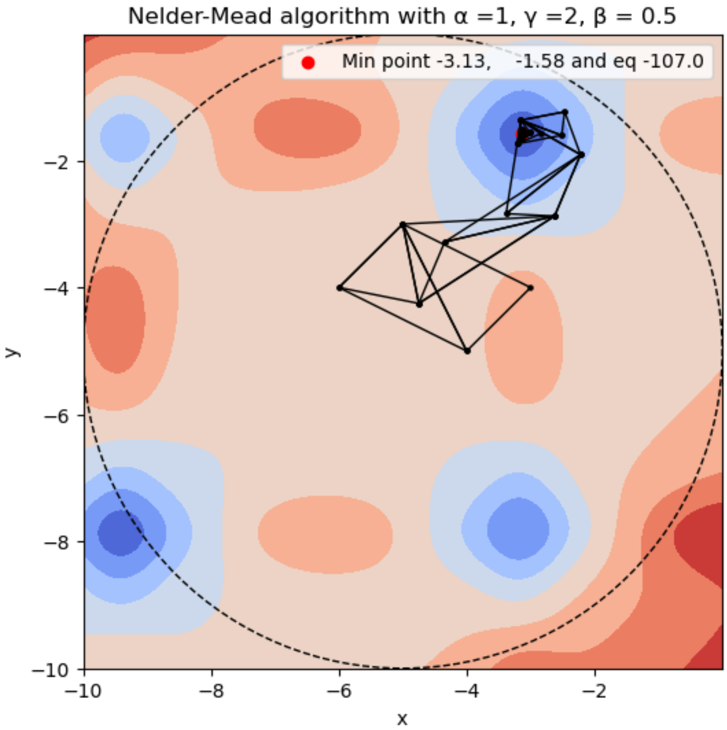
Table 2. Experiments with different hyperparameters for Nelder-Mead method

	α	γ	β
Case 1	1	2	0.5
Case 2	2	2	0.5
Case 3	1	4	0.5
Case 4	1	2	1.5
Case 5	3	2	3.5

In [9]:

```
simplex_diff_hp = [np.array([-3,-4]), np.array([-4,-5]), np.array([-5,-3])]

# case 1

x, f = nelder_mead_alg(simplex_diff_hp,
                       alpha = 1, gamma = 2, beta = 0.5)
```

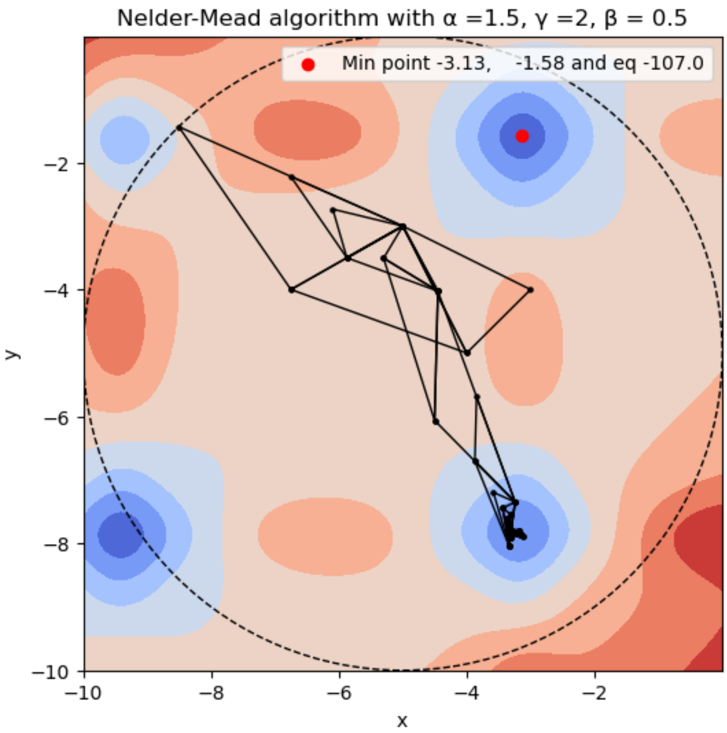
```
Nelder-Mead algorithm completed
x* = -3.1312062450387352, y* = -1.5811490956026952
f(x*, y*) = -106.76427293825802
Completed in 24 iterations
Oracle calls 119
```



In [10]:

```
# case 2
x, f = nelder_mead_alg(simplex_diff_hp,
                        alpha = 1.5, gamma = 2, beta = 0.5)
```

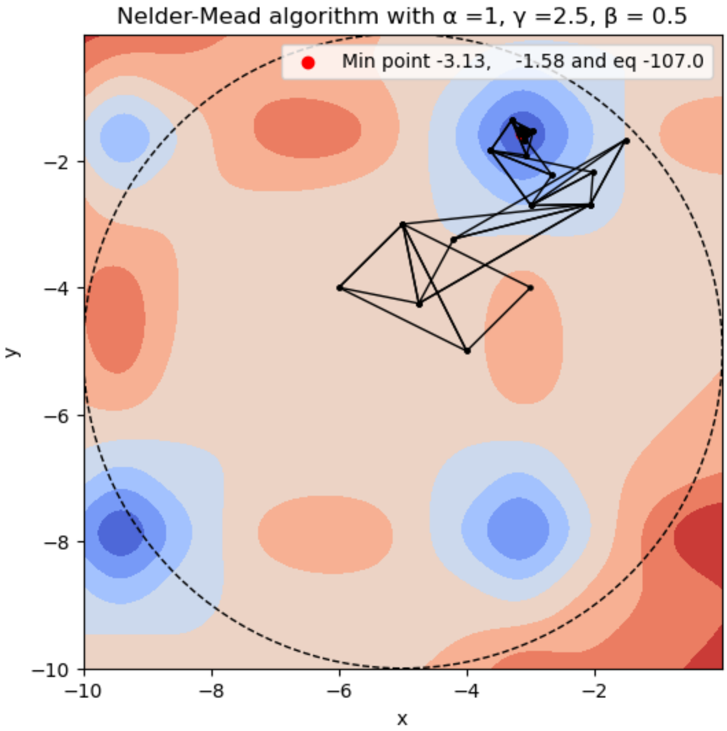
```
Nelder-Mead algorithm completed
x* = -3.1759216747071783, y* = -7.820231699902483
f(x*, y*) = -87.3108575377754
Completed in 33 iterations
Oracle calls 162
```



In [11]:

```
# case 3
x, f = nelder_mead_alg(simplex_diff_hp,
                       alpha = 1, gamma = 2.5, beta = 0.5)
```

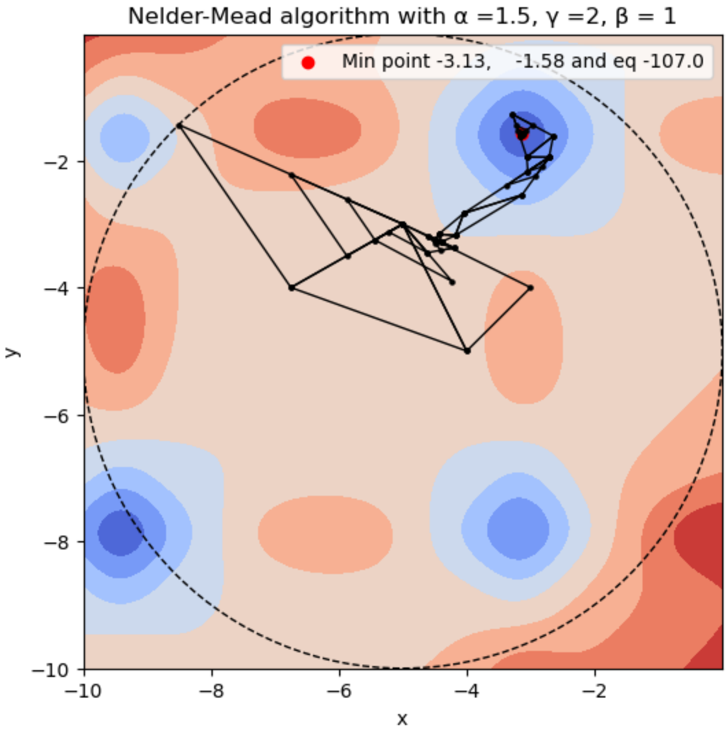
```
Nelder-Mead algorithm completed
x* = -3.1321444940585934, y* = -1.579163668819092
f(x*, y*) = -106.76281203123439
Completed in 23 iterations
Oracle calls 112
```



In [12]:

```
# case 4
x, f = nelder_mead_alg(simplex_diff_hp,
                          alpha = 1.5, gamma = 2, beta = 1)
```

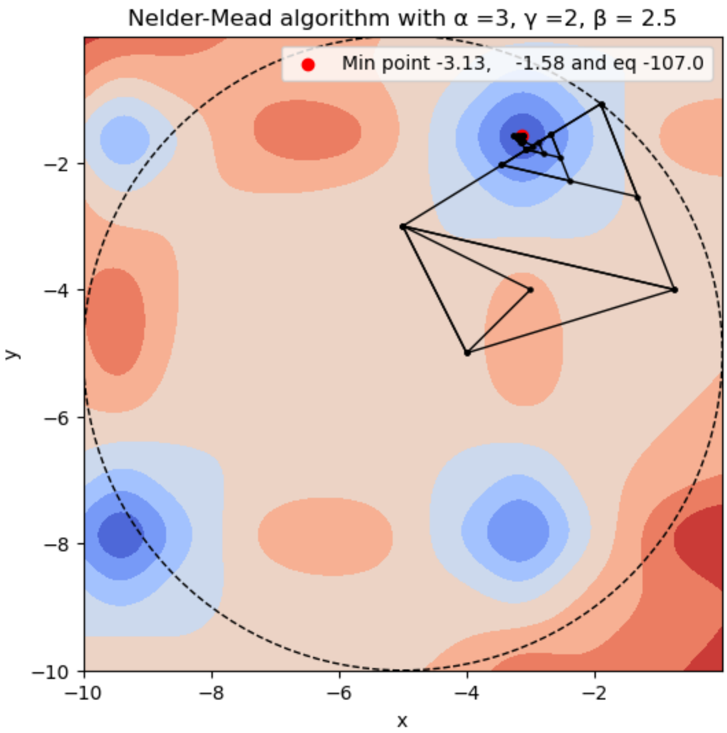
```
Nelder-Mead algorithm completed
x* = -3.131322506927109, y* = -1.5805493170299025
f(x*, y*) = -106.76402585656807
Completed in 28 iterations
Oracle calls 136
```



In [13]:

```
# case 5
x, f = nelder_mead_alg(simplex_diff_hp,
                       alpha = 3, gamma = 2, beta = 2.5)
```

```
Nelder-Mead algorithm completed
x* = -3.1302085610329207, y* = -1.5867233501991
f(x*, y*) = -106.76165368813318
Completed in 14 iterations
Oracle calls 70
```



4 Coordinate descent – 6 points

Implement coordinate descent for x^0 and f from Task 3. Compare the number of function evaluations (Oracle calls) for Nelder–Mead algorithm and coordinate descent. Report parameters of the algorithm. Attach your Jupyter notebook. Make a conclusion.

Solution

To compute the gradients, which will be used in Standard Coordinate descent, we can found them analytically

$$\begin{cases} \dfrac{\partial f}{\partial x} = 2(x-y) - \sin(x)(e^{(1-\sin(y))^2} - 2e^{(\cos(x)-1)^2}(1-\cos(x))\sin(y)) \\ \dfrac{\partial f}{\partial y} = \cos(y)(e^{(1-\cos(x))^2} - 2\cos(x)e^{(\sin(y)-1)^2}(1-\sin(y))) - 2x + 2y \end{cases} \tag{18}$$

In [14]:

```
def grad_calc_x(x,y):
    out = 2*(x-y) - np.sin(x)*(np.exp(np.power(1-np.sin(y),2)) - 2*np.exp(np.power(np.cos(x),2))*(1-np.cos(x))*np.sin(y
    return out
```

In [15]:

```
def grad_calc_y(x,y):
    out = np.cos(y)* (np.exp(np.power(1-np.cos(x),2)) - 2*np.cos(x)*np.exp(np.power(np.sin(y)-1,2))*(1-np.sin(y))-2*x+2
    return out
```

In [28]:

```
import sys
sys.setrecursionlimit(7000)

class StandardCoordinateDescent():

    def __init__(self, x, y, L, tol = 1e-6, check_method = 'norm_f', max_iter = 500):
        '''
        x,y in input - x0, y0
        L - hyperparameter
        tol - tolerance - error value for convergance
        check_method: norm_f, norm_xy, iter_num
        * norm_f: ||f(x_{k+1}) - f(x_k)||_2 < tol -> stop
        * norm_xy: ||x_{k+1} - x_k||_2 < tol -> stop
        * max_iter: current iteration achivied max_iter -> stop
        k - current iteration
        adaptive_alpha: alpha = alpha/k
        '''

        #initialization
        self.x = x
        self.y = y
        self.L = L
        self.tol = tol
        self.check_method = check_method
        self.max_iter = max_iter
        self.k = 0
        self.gamma = 1./self.L
        self.oracle = 0

        #grid settings for plotting
        xmin, xmax, step = -10, 0, 0.001
        xx = np.arange(xmin, xmax, step )
        yy = np.arange(xmin, xmax, step )
        xgrid, ygrid = np.meshgrid(xx, yy)
        zgrid = task3_objective(xgrid, ygrid)
        #plot setup
        fig = plt.figure(figsize=(6,6));
        ax2d = fig.add_subplot();
        ax2d.contourf(xgrid, ygrid, zgrid, cmap='coolwarm');
        circle = matplotlib.patches.Circle((-5,-5), radius=5, fill = False, ls = '--');
        ax2d.add_artist(circle);
        x_m, y_m, z_m = min_vals_searching(xgrid, ygrid, zgrid)
        ax2d.scatter(x_m,y_m,
                     label = f'Min point {np.round(x_m,2)},\
{np.round(y_m,2)} and eq {np.round(z_m)}', c='orange');
        ax2d.scatter(x,y,
                     label = f'Initial point {x},\
{y}', c='blue');
        ax2d.set_xlabel('x')
        ax2d.set_ylabel('y')
        plt.title(f'Standard Coord Descent with L ={self.L}')
        plt.legend();


    def obj_function(self):
        self.obj = task3_objective(self.x, self.y)
        self.oracle += 1


    def grad_calc(self):
        self.obj_function()
        self.grad_x =  grad_calc_x(self.x, self.y)
        self.grad_y =  grad_calc_y(self.x, self.y)
        self.oracle += 1


    def step(self):
        self.grad_calc()
        self.obj_prev = task3_objective(self.x, self.y)
        self.x_prev = self.x
        self.y_prev = self.y
        self.x = self.x - self.gamma * self.grad_x
        self.y = self.y - self.gamma * self.grad_y
        self.obj_function()
        self.k += 1
        self.plot()

    def check(self):
        self.step()
        if self.check_method == 'norm_f':
            if np.linalg.norm([self.obj - self.obj_prev],2) < self.tol or self.k>13000:
                self.end = True
            else:
                self.end= False
        elif self.check_method == 'norm_xy':
            if np.linalg.norm(np.array([self.x, self.y]) - np.array([self.x_prev, self.y_prev]),2) < self.tol \
            or self.k>13000:
```

```
                self.end = True
            else:
                self.end= False
        else:
            if self.k > self.max_iter:
                self.end = True
            else:
                self.end= False

    def plot(self):
        plt.arrow(self.x_prev, self.y_prev,self.x-self.x_prev, self.y-self.y_prev,
                  color = 'red', width = 0.01)

    def result(self):
        self.obj_function()
        print('Standard coordinate descent completed')
        print(f'x* = {self.x}, y* = {self.y}')
        print(f'f(x*, y*) = {self.obj}')
        print(f'Completed in {self.k} iterations')
        print(f'Calling the oracle {self.oracle} times')
        plt.show()

    def descent(self):
        self.check()
        if self.end:
            self.result()
        else:
            self.step()
            self.descent()
```

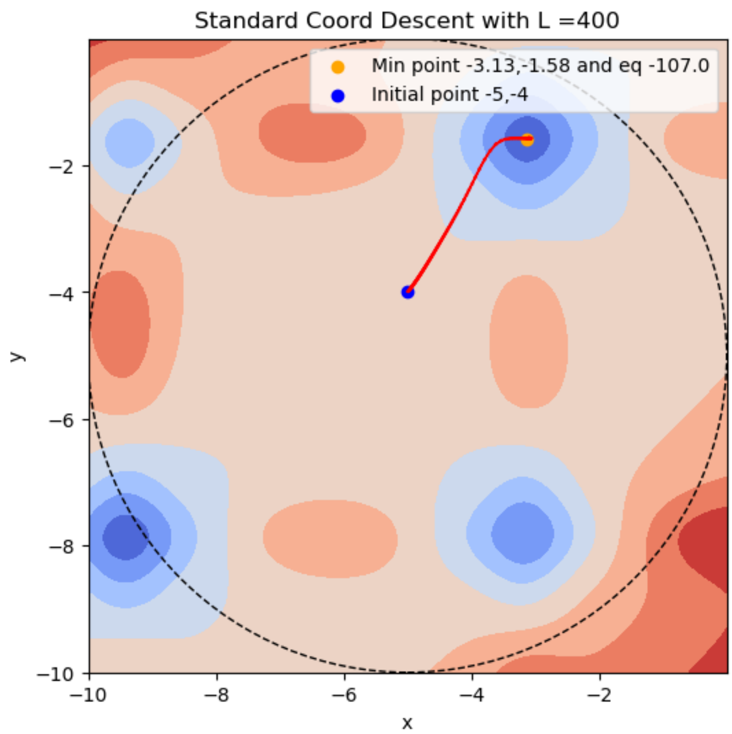
Table 3. Experiments with different initial points for Standard Coordinate Descent

Cases	x	y
Case 1	-5	-4
Case 2	-5	-7
Case 3	-2	-5
Case 4	-9	-5

In [29]:

```
# case 1

birdSCD1 = StandardCoordinateDescent(x = -5, y = -4, L = 400, tol = 1e-4)
birdSCD1.descent()
```

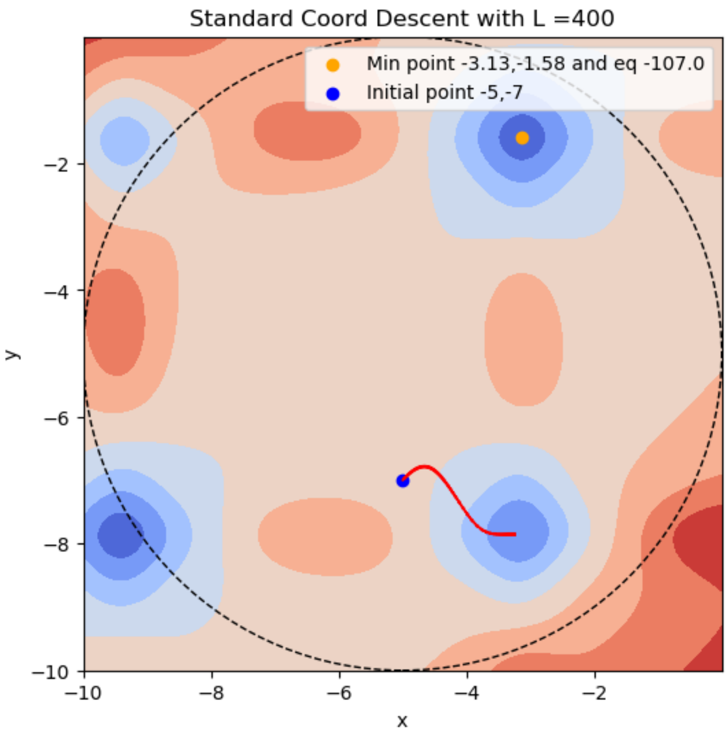
```
Standard coordinate descent completed
x* = -3.0950399840350213, y* = -1.5707963267948968
f(x*, y*) = -106.57780438841314
Completed in 157 iterations
Calling the oracle 472 times
```



In [30]:

```
# case 2

birdSCD2 = StandardCoordinateDescent(x = -5, y = -7, L = 400, tol = 1e-4)
birdSCD2.descent()
```

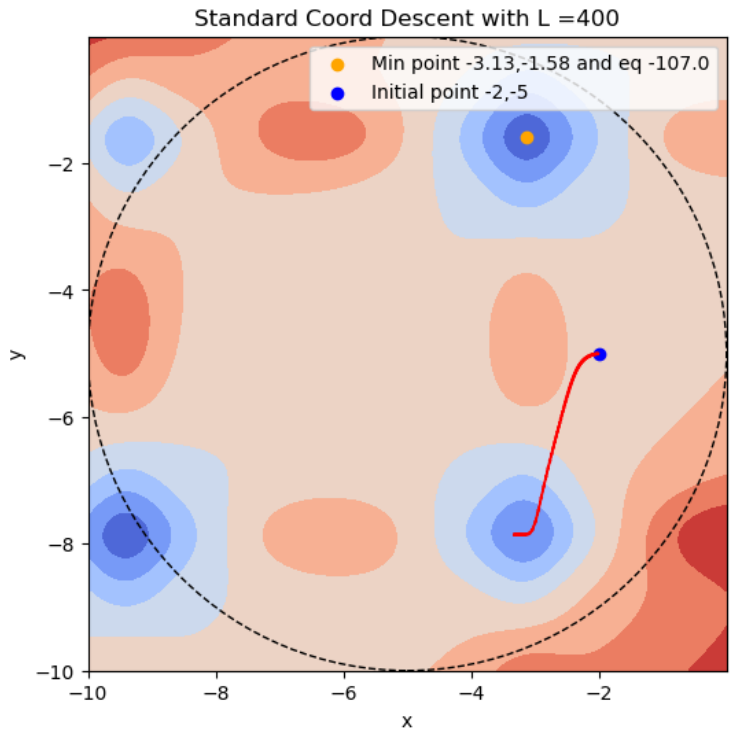
```
Standard coordinate descent completed
x* = -3.282293494489466, y* = -7.853981633974483
f(x*, y*) = -85.64548393055716
Completed in 105 iterations
Calling the oracle 316 times
```



In [31]:

```
# case 3

birdSCD3 = StandardCoordinateDescent(x = -2, y = -5, L = 400, tol = 1e-4)
birdSCD3.descent()
```

```
Standard coordinate descent completed
x* = -3.2822608815505157, y* = -7.853981633974483
f(x*, y*) = -85.64639078068697
Completed in 133 iterations
Calling the oracle 400 times
```



In [32]:

```
# case 4

birdSCD4 = StandardCoordinateDescent(x = -9, y = -5, L = 400, tol = 1e-6, max_iter=45, check_method = 'max_iter' )
birdSCD4.descent()
```

```
Standard coordinate descent completed
x* = -8.97932153861403, y* = -7.815121266401395
f(x*, y*) = -85.04501691762844
Completed in 47 iterations
Calling the oracle 142 times
```

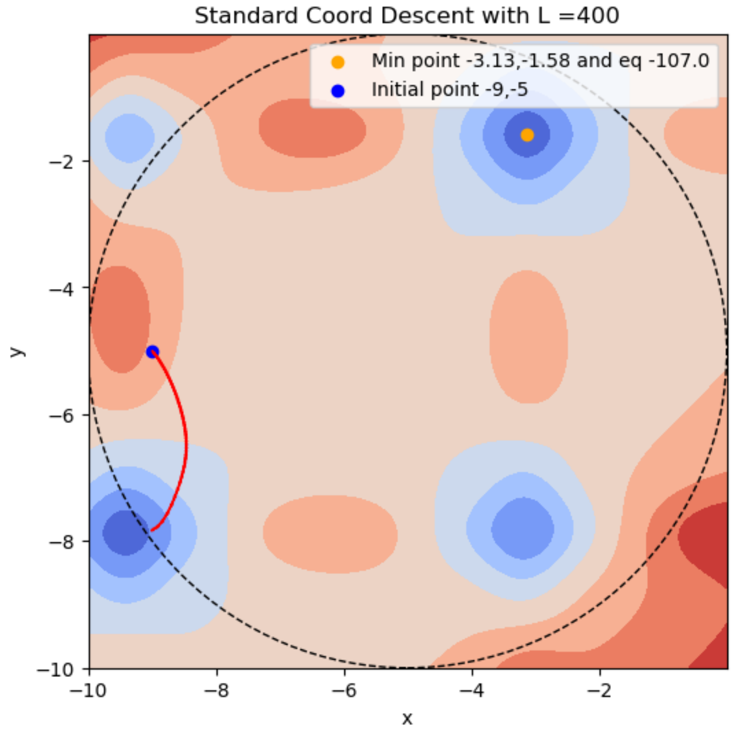


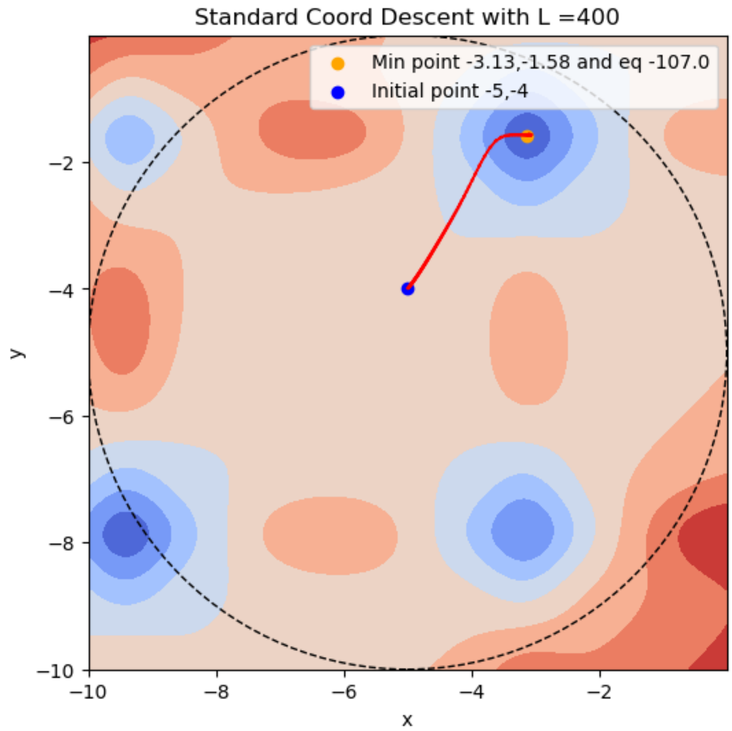
Table 2. Experiments with different hyperparameters for Standard Coordinate Descent

Hyperparameter	Case 1	Case 2	Case 3
L	400	100	1000

In [33]:

```
birdSCD5 = StandardCoordinateDescent(x = -5, y = -4, L = 400, tol = 1e-4)
birdSCD5.descent()
```

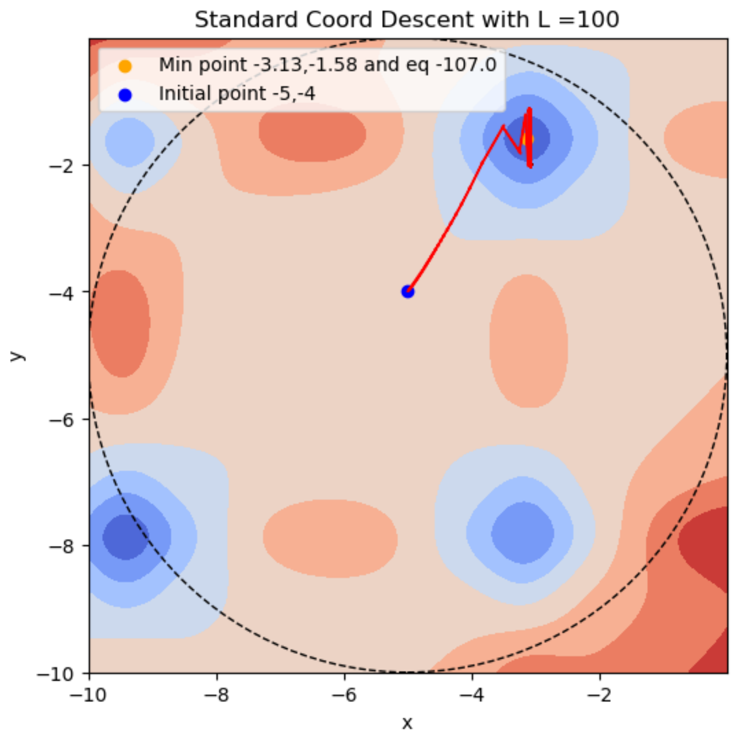
```
Standard coordinate descent completed
x* = -3.0950399840350213, y* = -1.5707963267948968
f(x*, y*) = -106.57780438841314
Completed in 157 iterations
Calling the oracle 472 times
```



In [34]:

```
birdSCD6 = StandardCoordinateDescent(x = -5, y = -4, L = 100, tol = 1e-4)
birdSCD6.descent()
```

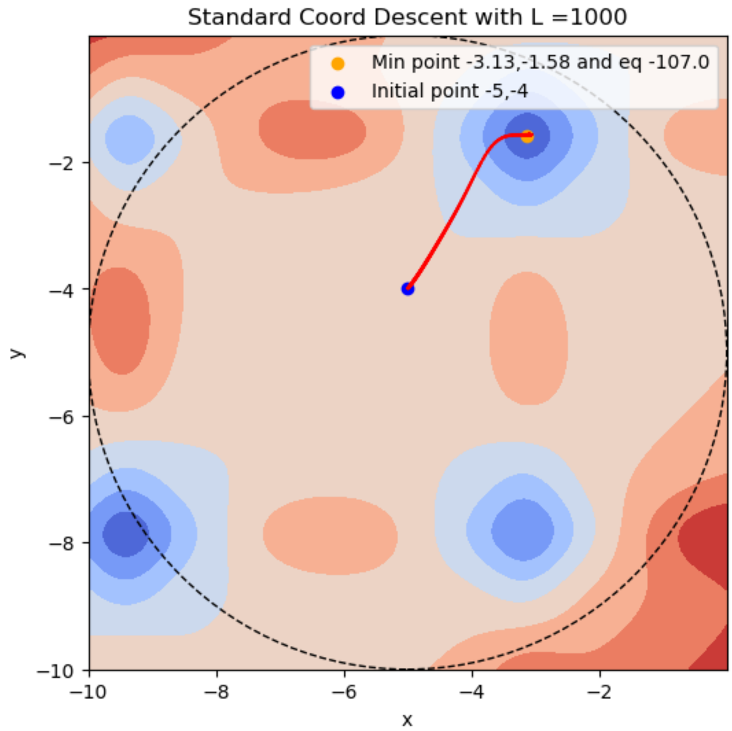
```
Standard coordinate descent completed
x* = -3.07444037023142, y* = -1.9941125248645317
f(x*, y*) = -86.73474021542978
Completed in 13001 iterations
Calling the oracle 39004 times
```



In [35]:

```
birdSCD7 = StandardCoordinateDescent(x = -5, y = -4, L = 1000, tol = 1e-5)
birdSCD7.descent()
```

```
Standard coordinate descent completed
x* = -3.0950075494447122, y* = -1.570796326794897
f(x*, y*) = -106.57749268038162
Completed in 415 iterations
Calling the oracle 1246 times
```



In [36]:

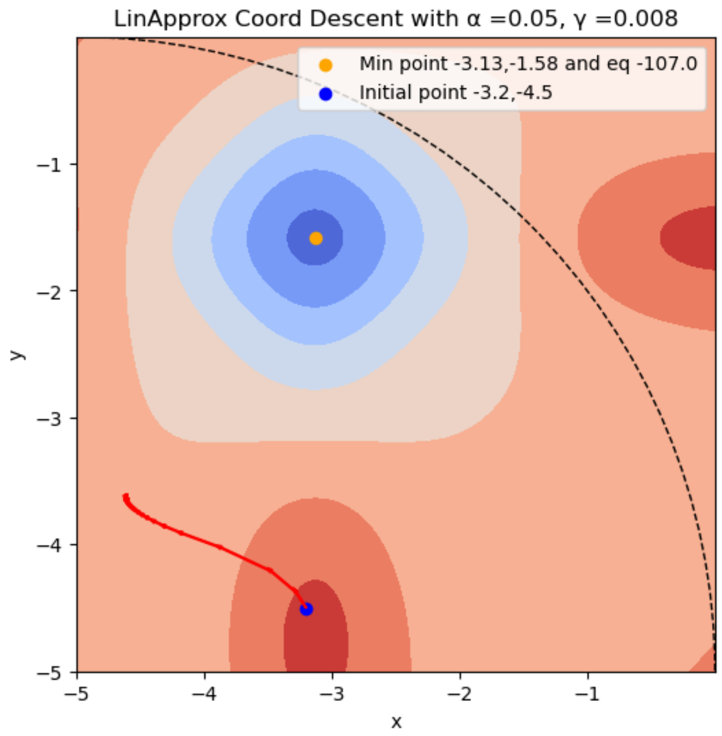
```
import sys
sys.setrecursionlimit(7000)

class LinApproxCoordinateDescent():

    def __init__(self, x, y, alpha, gamma, tol, adaptive_alpha = False,
                 check_method = 'norm_f', max_iter = 500):
        '''
        x,y in input - x0, y0
        alpha, gamma - hyperparameters
        tol - tolerance - error value for convergance
        check_method: norm_f, norm_xy, iter_num
        * norm_f: ||f(x_{k+1}) - f(x_k)||_2 < tol -> stop
        * norm_xy: ||x_{k+1} - x_k||_2 < tol -> stop
        * max_iter: current iteration achivied max_iter -> stop
        k - current iteration
        adaptive_alpha: alpha = alpha/k
        '''

        #initialization
        self.x = x
        self.y = y
        self.alpha = alpha
        self.gamma = gamma
        self.tol = tol
        self.adaptive_alpha = adaptive_alpha
        self.check_method = check_method
        self.max_iter = max_iter
        self.k = 0
        self.oracle = 0

        #grid settings for plotting
        xmin, xmax, step = -5, 0, 0.001
        xx = np.arange(xmin, xmax, step )
        yy = np.arange(xmin, xmax, step )
        xgrid, ygrid = np.meshgrid(xx, yy)
        zgrid = task3_objective(xgrid, ygrid)
        #plot setup
        fig = plt.figure(figsize=(6,6));
        ax2d = fig.add_subplot();
        ax2d.contourf(xgrid, ygrid, zgrid, cmap='coolwarm');
        circle = matplotlib.patches.Circle((-5,-5), radius=5, fill = False, ls = '--');
        ax2d.add_artist(circle);
        x_m, y_m, z_m = min_vals_searching(xgrid, ygrid, zgrid)
        ax2d.scatter(x_m,y_m,
                     label = f'Min point {np.round(x_m,2)},\
{np.round(y_m,2)} and eq {np.round(z_m)}', c='orange');
        ax2d.scatter(x,y,
                     label = f'Initial point {x},\
{y}', c='blue');
        ax2d.set_xlabel('x')
        ax2d.set_ylabel('y')
        plt.title(f'LinApprox Coord Descent with α ={self.alpha}, γ ={self.gamma}')
        plt.legend();


    def obj_function(self):
        self.obj_alpha_x = task3_objective(self.x+self.alpha, self.y)
        self.obj_alpha_y = task3_objective(self.x, self.y+self.alpha)
        self.obj = task3_objective(self.x, self.y)
        self.oracle += 3



    def s_calc(self):
        self.obj_function()
        self.s_x =  1./self.alpha * (self.obj_alpha_x - self.obj)
        self.s_y =  1./self.alpha * (self.obj_alpha_y - self.obj)


    def step(self):
        self.s_calc()
        self.obj_prev = task3_objective(self.x, self.y)
        self.x_prev = self.x
        self.y_prev = self.y
        self.x = self.x - self.gamma * self.s_x
        self.y = self.y - self.gamma * self.s_y
        self.k += 1
        if self.adaptive_alpha and self.k<100:
            self.alpha = self.alpha/self.k
        self.plot()

    def check(self):
        self.step()
        if self.check_method == 'norm_f':
            if np.linalg.norm([self.obj - self.obj_prev],2) < self.tol or self.k>13000:
                self.end = True
```

```
            else:
                self.end= False
        elif self.check_method == 'norm_xy':
            if np.linalg.norm(np.array([self.x, self.y]) - np.array([self.x_prev, self.y_prev]),2) < self.tol \
            or self.k>13000:
                self.end = True
            else:
                self.end= False
        else:
            if self.k > self.max_iter:
                self.end = True
            else:
                self.end= False

    def plot(self):
        plt.arrow(self.x_prev, self.y_prev,self.x-self.x_prev, self.y-self.y_prev,
                 color = 'red', width = 0.01)

    def result(self):
        self.obj_function()
        print('Linear Approximation coordinate descent completed')
        print(f'x* = {self.x}, y* = {self.y}')
        print(f'f(x*, y*) = {self.obj}')
        print(f'Completed in {self.k} iterations')
        print(f'Calling the oracle {self.oracle} times')
        plt.show()

    def descent(self):
        self.check()
        if self.end:
            self.result()
        else:
            self.step()
            self.descent()
```

In [37]:

```
birdLACD = LinApproxCoordinateDescent(x = -3.2,y = -4.5, alpha = 0.05, gamma = 0.008, tol = 1e-5,
                                       adaptive_alpha=True, check_method='max_iter', max_iter=13000)
birdLACD.descent()
```

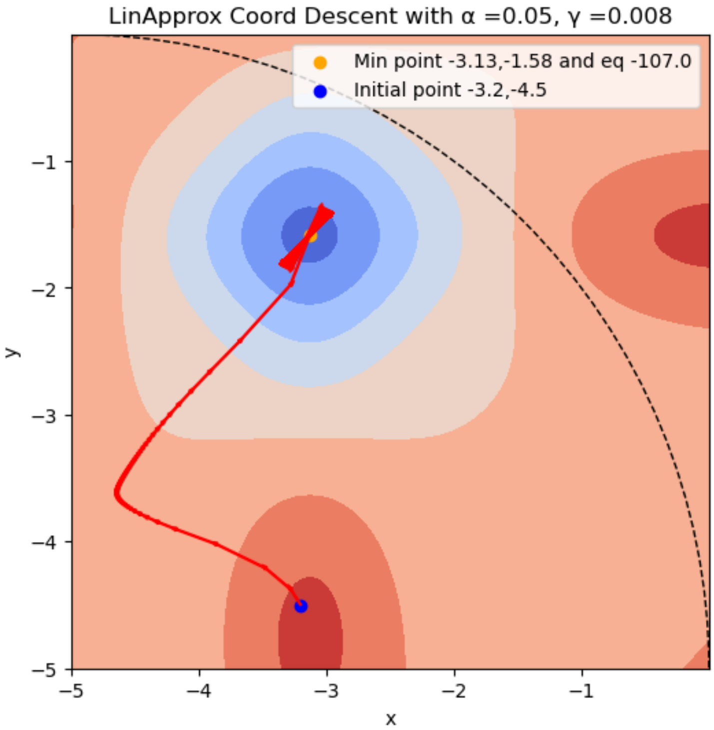
```
Linear Approximation coordinate descent completed
x* = -4.6086113982182, y* = -3.648338885743031
f(x*, y*) = 2.427610201201521
Completed in 13001 iterations
Calling the oracle 39006 times
```



In [39]:

```
birdLACD1 = LinApproxCoordinateDescent(x = -3.2,y = -4.5, alpha = 0.05, gamma = 0.008, tol = 1e-5,
                                       adaptive_alpha=False, check_method='max_iter', max_iter=13000)
birdLACD1.descent()
```

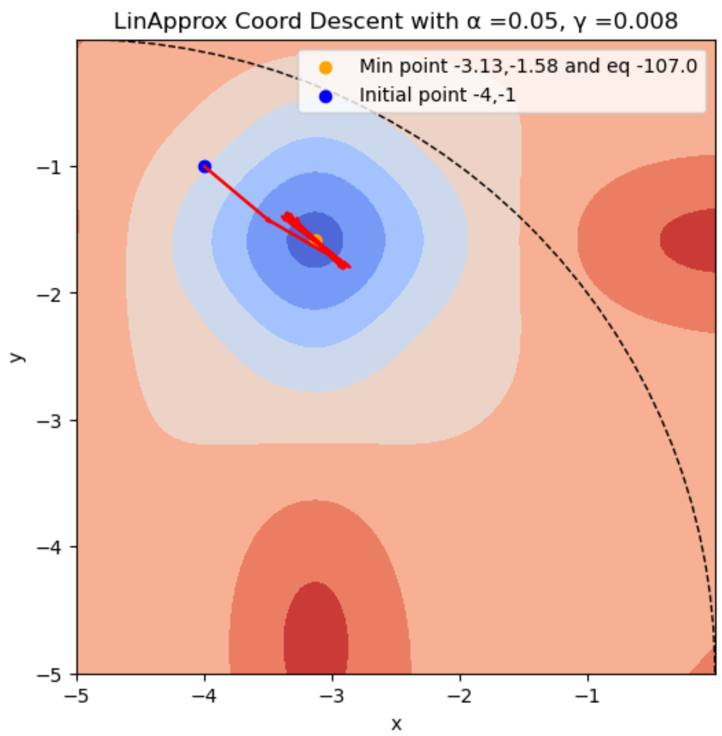
```
Linear Approximation coordinate descent completed
x* = -3.3317726820352305, y* = -1.7861318623748608
f(x*, y*) = -96.16196006811784
Completed in 13001 iterations
Calling the oracle 39006 times
```



In [38]:

```
birdLACD2 = LinApproxCoordinateDescent(x = -4,y = -1, alpha = 0.05, gamma = 0.008, tol = 1e-5, adaptive_alpha=True,
                        check_method='norm_xy', max_iter=1000)
birdLACD2.descent()
```

```
Linear Approximation coordinate descent completed
x* = -3.269398009874679, y* = -1.4494854163249724
f(x*, y*) = -101.73566929847519
Completed in 19 iterations
Calling the oracle 60 times
```



Conclusion

Nelder-Mead method

1. Initial simplex

The initial simplex is critical to the success of the Nelder-Mead algorithm. If it is too small, the search can become localized, which can cause the algorithm to get trapped. The size of the simplex should be relevant to the problem at hand. The article in question suggests using a simplex with an initial point of $\mathbf{x_1}$ and the other points generated with a fixed step along each dimension. This means that the method is sensitive to the scaling of the variables that make up $\mathbf{x}$.

The algorithm can converge to different local minima depending on the area to which the simplex points belong.

2. Hyperparameters (α, γ, β)

Each of the hyperparameters is important in calculating the algorithm. A drastic change in the α parameter in the second case of the experiment with a fixed initial simplex led to finding an incorrect local minimum.

Standard Coordinate

1. Initial point

As with the Nelder-Mead method, the starting point is also extremely important, because different values of the starting point give approximation to different local minima, instead of converging to a single global minimum.

2. Hyperparameter (L)

The training parameter significantly affects the number of iterations: the larger the parameter is, the smaller the gradient increment will be, and the slower the convergence to the minimum will be. But for all that, if parameter L is too small (i.e. the gradient increment is too large), then the algorithm's endpoint will volatilize near the minimum, not getting into it in any way (which was demonstrated in case 3 at L=100).

Comparing Comparing these two algorithms we would like to note that:

1. Nelder-Mead does not require to know the function explicitly because it does not require a differentiation operation to calculate the gradient.
2. Nelder-Mead converges faster than Standrate Coordinate Descent (usually tens of operations vs. hundreds of operations).
3. Nelder-Mead requires multiple times as many Oracle function calls per operation (Nelder-Mead requires 5 function calls vs. 3 function calls for Standard Coordinate Descent.

Linear Approximation Coordinate

1. Initial point

This method proved to be the most sensitive to the starting point. If the starting point is close to the initial minimum, the method performs well, and does so in fewer operations than the Standard Coordinate Descent. Otherwise, the algorithm may get stuck in the plane of the function (if the value of the function changes little in any part of it).

2. Hyperparameter (α, γ)

Whether or not the α parameter adapts makes the algorithm more sensitive to the distance of the starting point (in the second case we see how eliminating the α parameter adaptation made it possible to overcome the trap I wrote about above.

Comparing It is similar to Standard Coordinate Descent in terms of the cost of the Oracle function, but it has the advantage of the Nelder-Mead algorithm: it does not need to know the function explicitly in order to find the minimum. But it is less accurate than the Standard Coordinate Descent.

Appendix: Not working class code for the Nelder-Mead algorithm

In [27]:

```
#в разработке, код не работает, смотрите выше

import sys
sys.setrecursionlimit(7000)

class NelderMead():

    def __init__(self, init_simplex, alpha=1, beta = 0.5, gamma=2, tol = 1e-2, max_iter = 5000,
                check_method = 'norm'):
        self.init_simplex = init_simplex
        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma
        self.tol = tol
        self.k = 0
        self.max_iter = max_iter
        self.check_method = check_method

    def process(self):
        self.sort()
        self.k += 1
        self.centroid()
        self.reflection()
        self.control()
        self.comparsion()

    def comparsion(self):
        if self.f_r < self.f_l:
            self.a4()
        elif self.f_l < self.f_r and self.f_r < self.f_g:
            self.b4()
        elif self.f_h > self.f_r and self.f_r > self.f_g:
            self.c4()
        else:
            self.d4()

    def a4(self):
        self.expansion()
        if self.f_e<self.f_l:
            buf = self.x_h
            self.x_h = self.x_e
            self.x_e = buf
            self.convergence()
        else:
            buf = self.x_h
            self.x_h = self.x_r
            self.x_r = buf
            self.convergence()

    def b4(self):
        buf = self.x_h
        self.x_h = self.x_r
        self.x_r = buf
        self.convergence()

    def c4(self):
        buf = self.x_h
        self.x_h = self.x_r
        self.x_r = buf
        self.d4()

    def d4(self):
        self.contraction()
        if self.f_s < self.f_h:
            buf = self.x_h
            self.x_h = self.x_s
            self.x_s = buf
            self.convergence()
        else:
            self.buf_arr = self.init_simplex
            self.init_simplex = []

            for item in self.buf_arr:
                if (item != self.x_l).any():
                    item = np.array(self.x_l) + 0.5*(np.array(item) - np.array(self.x_l))
                    self.init_simplex.append(item.tolist())
            self.convergence()

    def pre_processing(self):
        self.func_vals_init = []
        for item in self.init_simplex:
            self.func_vals_init.append(task3_objective(item[0], item[1]))

    def sort(self):
        self.pre_processing()
        self.f_h = np.max(self.func_vals_init)
        self.f_l = np.min(self.func_vals_init)
```

```python

        for item in self.func_vals_init:
            if item != self.f_h and item != self.f_l:
                self.f_g = item

        for i in range(len(self.func_vals_init)):
            if self.func_vals_init[i] == self.f_h:
                self.x_h = self.init_simplex[i]
            elif self.func_vals_init[i] == self.f_g:
                self.x_g = self.init_simplex[i]
            else:
                self.x_l = self.init_simplex[i]
        self.x_l = np.array(self.x_l)
        self.x_g = np.array(self.x_g)
        self.x_h = np.array(self.x_h)

    def centroid(self):
        self.x_c = 0.5*np.sum([self.x_g, self.x_l],axis=0)
#         self.x_c = self.x_c.tolist()

    def reflection(self):
#         self.x_r = (1+self.alpha) * np.array(self.x_c) + self.alpha*np.array(self.x_h)
#         self.x_r = self.x_r.tolist()
        self.x_r = (1+self.alpha) * self.x_c + self.alpha*self.x_h
        self.f_r = task3_objective(self.x_r[0], self.x_r[1])

    def expansion(self):
#         self.x_e = np.array(self.x_c)*(1+self.gamma) + self.gamma*np.array(self.x_h)
#         self.x_e = self.x_e.tolist()
        self.x_e = self.x_c*(1+self.gamma) + self.gamma*self.x_h
        self.f_e = task3_objective(self.x_e[0], self.x_e[1])

    def contraction(self):
#         self.x_s = np.array(self.x_c)*(1+self.beta) + self.beta*np.array(self.x_h)
#         self.x_s = self.x_s.tolist()
        self.x_s = self.x_c *(1+self.beta) + self.beta* self.x_h
        self.f_s = task3_objective(self.x_s[0], self.x_s[1])

    def simplex_update(self):
        self.init_simplex =[]
        self.init_simplex.append(self.x_l)
        self.init_simplex.append(self.x_g)
        self.init_simplex.append(self.x_h)

    def convergence(self):
        if self.check_method == 'variance':
            self.convergence_var()
        elif self.check_method == 'norm':
            self.convergence_norm()
        else:
            if self.k >= self.max_iter:
                self.result()
            else:
                self.simplex_update()
                self.process()

    def convergence_norm(self):
        self.norm_diff = np.linalg.norm(self.x_l - self.x_h,2)
        if self.norm_diff < self.tol or self.k > self.max_iter:
            self.result()
        else:
            self.simplex_update()
            self.process()

    def convergence_var(self):
        self.var_x = np.var(np.array([self.x_l[0], self.x_g[0], self.x_h[0]]))
        self.var_y = np.var(np.array([self.x_l[1], self.x_g[1], self.x_h[1]]))
        self.var = self.var_x+self.var_y
        if self.var < self.tol or self.k >= self.max_iter:
            self.result()
        else:
            self.simplex_update()
            self.process()


    def control(self):
        print(f'f_h = {self.f_h}, f_g = {self.f_g}, f_l = {self.f_l}')
        print(f'x_h = {self.x_h}, x_g = {self.x_g}, x_l = {self.x_l}')
        print(f'x_c = {self.x_c}')
        print(f'x_r = {self.x_r},  f_r = {self.f_r}')
#         print(f'x_e = {self.x_e},  f_e = {self.f_e}')

    def result(self):
        print(f'f_h = {self.f_h}, f_g = {self.f_g}, f_l = {self.f_l}')
        print(f'x_h = {self.x_h}, x_g = {self.x_g}, x_l = {self.x_l}')
        print(f'k = {self.k}')

```