Alexa Tartaglini's solutions to problem 2

November 7, 2022

1 Alexa Tartaglini's solutions to problem 2

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[59]:
```
# Define the relevant functions

def LJ_potential(r, sigma=1, epsilon=1):
    '''
    This function computes the Lennard-Jones potential between two particles.

    :param r: the distance between the particles.
    :param sigma: the sigma parameter (scalar)
    :param epsilon: the epsilon parameter (scalar)

    :return: the Lennard-Jones potential (scalar)
    '''
    return 4 * epsilon * ((sigma / r)**12 - (sigma/r)**6)

def f(P):
    '''
    This function computes the total potential energy for a system of␣
↪particles, P.

    :param P: the particle system (mx2 matrix)

    :return: the total potential energy for the system (scalar)
    '''
    te = 0

    for i in range(P.shape[0]):
        for j in range(P.shape[0]):
            if i == j:
                continue

            r_ij = np.linalg.norm(np.subtract(P[i, :], P[j, :]))
```

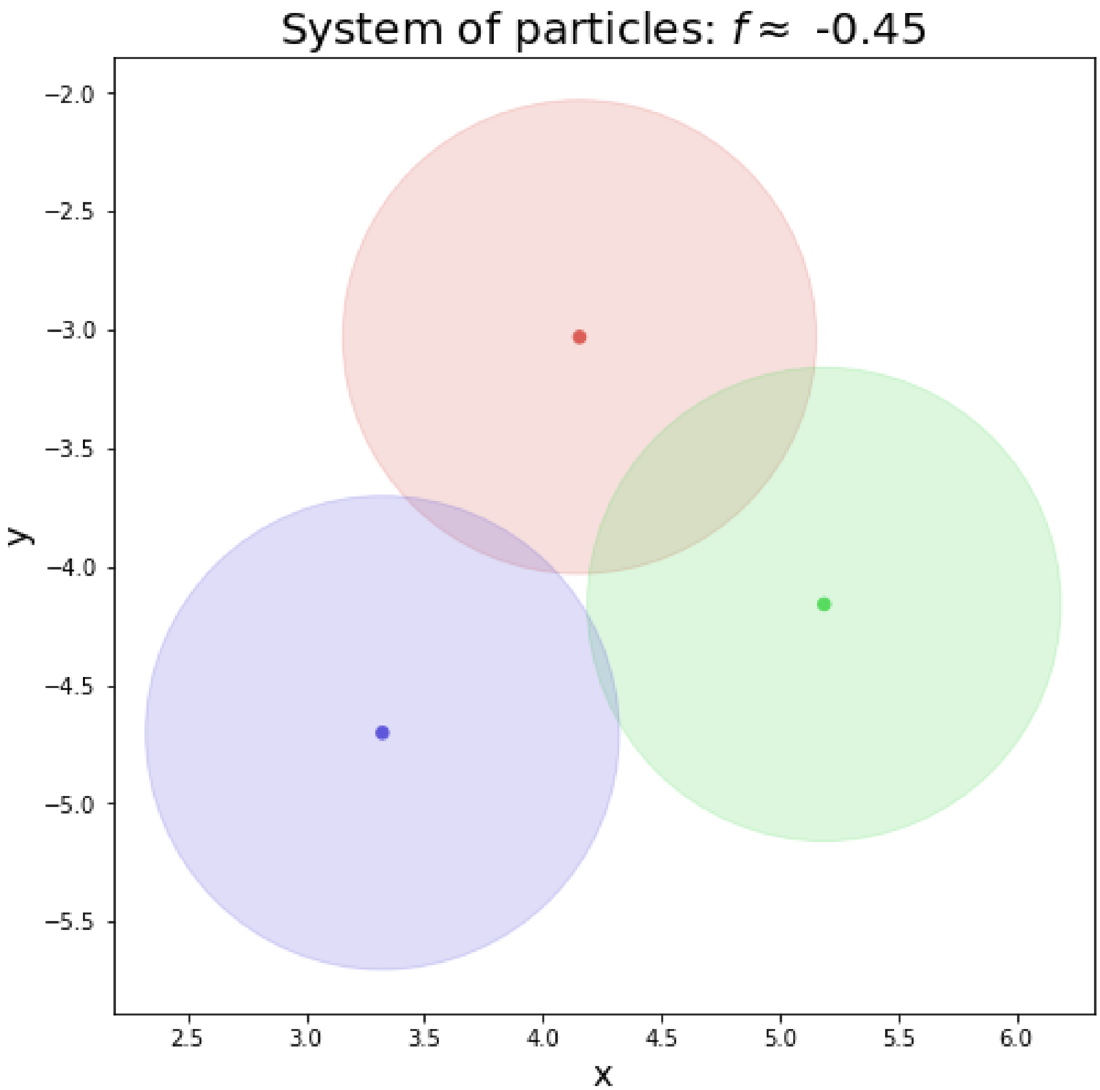
```
            te += LJ_potential(r_ij)

    return te / 2

def grad_f(P):
    '''
    This function computes the gradient of f evaluated at current particle␣
 ↪positions P.

    :param P: the particle system (mx2 matrix)

    :return: the gradient of f evaluated at P (2mx1 vector)
    '''
    grad = np.zeros((2 * P.shape[0], 1))
    grad_idx = 0

    for i in range(P.shape[0]):  # Iterate over points in P
        pi = P[i, :]
        xi = pi[0]
        yi = pi[1]

        for j in range(P.shape[0]):
            if i == j:
                continue

            pj = P[j, :]
            xj = pj[0]
            yj = pj[1]

            norm = np.linalg.norm(pi - pj)

            grad[grad_idx] += 4*(-12 * (xi - xj) / norm**14) + 6*(xi - xj) /␣
 ↪norm**8
            grad[grad_idx + 1] += 4*(-12 * (yi - yj) / norm**14) + 6*(yi - yj) /
 ↪ norm**8

        grad_idx += 2

    return grad / 2

def back_tracking(alpha, beta, P, direction):
    '''
    Backtracking line search.
    '''
    te = f(P)

    P_next = P + alpha*direction
```

```
    te_next = f(P_next)  # want Fn_1 < Fn

    while te < te_next:
        alpha = alpha*beta
        P_next = P + alpha*direction
        te_next = f(P_next)

    return alpha

def BFGS(P0, H0, maxIter=5000, tol=1e-12, alpha0=1, beta=0.8):
    '''
    This function minimizes f for an initial system of particles P0 and initial␣
 ↪inverse Hessian approximation H0
    using the BFGS update to the Hessian.

    :param P0: initial particle system (mx2 matrix)
    :param H0: initial inverse Hessian approximation (2mx2m matrix)
    :param maxIter: maximum number of iterations allowed
    :param tol: tolerance; if the norm of the gradient is below this, then stop
    :param alpha0: initial alpha for backtracking
    :param beta: backtracking parameter (alpha_k+1 = alpha_k * beta)

    :return:
    '''
    P_vec = [P0]  # Stores the system of particles P at each iteration
    f_vec = [f(P0)]  # Stores the total potential energy at each iteration
    i = 0  # iteration counter

    Hk = H0
    Pn = P0
    g = grad_f(Pn)

    rho_inv = np.ones(g.shape)
    I = np.identity(H0.shape[0])

    while i < maxIter and np.linalg.norm(g) >= tol and np.linalg.norm(rho_inv)␣
 ↪>= tol:
        direction = (-Hk @ g).reshape(-1, 2)  # reshape 2mx1 -> mx2 to make␣
 ↪addition with P easier
        alpha = back_tracking(alpha0, beta, Pn, direction)

        Pn = Pn + alpha*direction

        g_next = grad_f(Pn)
        sk = alpha*direction.reshape(2*P0.shape[0], 1)
        yk = g_next - g  # yk = change in gradient.
```

```
        Hk = (I - 1/rho_inv*sk@sk.T)@Hk@(I - 1/rho_inv*yk@sk.T) + 1/
↪rho_inv*sk@sk.T
        P_vec.append(Pn)
        f_vec.append(f(Pn))
        g = g_next
        i += 1

    return Pn, P_vec, f_vec, i

def plot_particles(ax, P, te, distances=False):
    '''
    Creates a scatter plot of the particle system P.

    :param ax: Axes object to create the plot on
    :param P: the system of particles (mx2 matrix)
    :param te: f achieved for this system.
    :param distances: True if the distances between particles should be drawn.
    '''
    c = sns.color_palette('hls', P.shape[0])  # distinct colors for each␣
↪particle
    xs = []
    ys = []

    for i in range(P.shape[0]):
        xs.append(P[i, 0])
        ys.append(P[i, 1])

    ax.scatter(xs, ys, c=c, s=30, label='Particles', marker='o', zorder=1)

    # radii
    for i in range(len(xs)):
        patch = plt.Circle((xs[i], ys[i]), 1, color=c[i], alpha=0.2)
        ax.add_patch(patch)

    ax.set_xlabel(r'x', fontsize=16)
    ax.set_ylabel(r'y', fontsize=16)
    ax.set_title(r'System of particles: $f \approx$ {}'.format(round(te, 2)),␣
↪fontsize=20)
    ax.axis('equal')

def init_P(m, dist_param=2):
    '''
    This function initializes the system of particles P such that the particles␣
↪are not too close together.

    :param m: the number of particles.
    '''
```

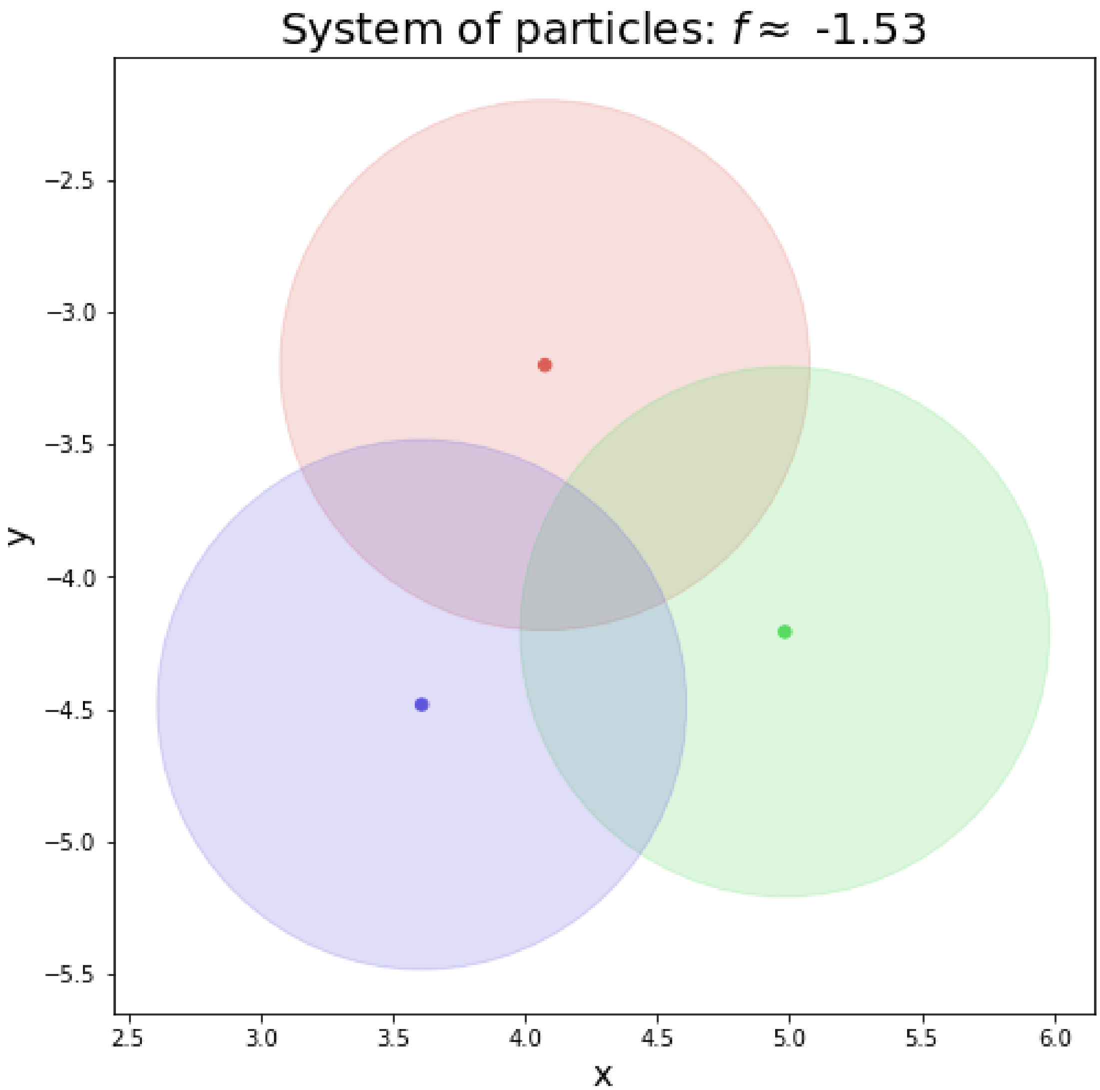
```
    P = np.zeros((m, 2))
    ps = []
    ps.append(np.random.uniform(low=-10, high=10, size=(2, 1)))

    tries = 0

    while len(ps) < m:
        if tries == 100:
            ps = []
            tries = 0

        p = np.random.uniform(low=-10, high=10, size=2)
        add = True

        for q in ps:
            if np.linalg.norm(np.subtract(p, q)) < 1.5 or np.linalg.norm(np.
↪subtract(p, q)) > dist_param:
                add = False
                break

        if add:
            ps.append(p)

        tries += 1

    for i in range(m):
        P[i, 0] = ps[i][0]
        P[i, 1] = ps[i][1]

    return P
```

[53]:
```
# Run the algorithm with m = 3
m = 3

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_3 = init_P(m)
plot_particles(ax, P0_3, f(P0_3))
```

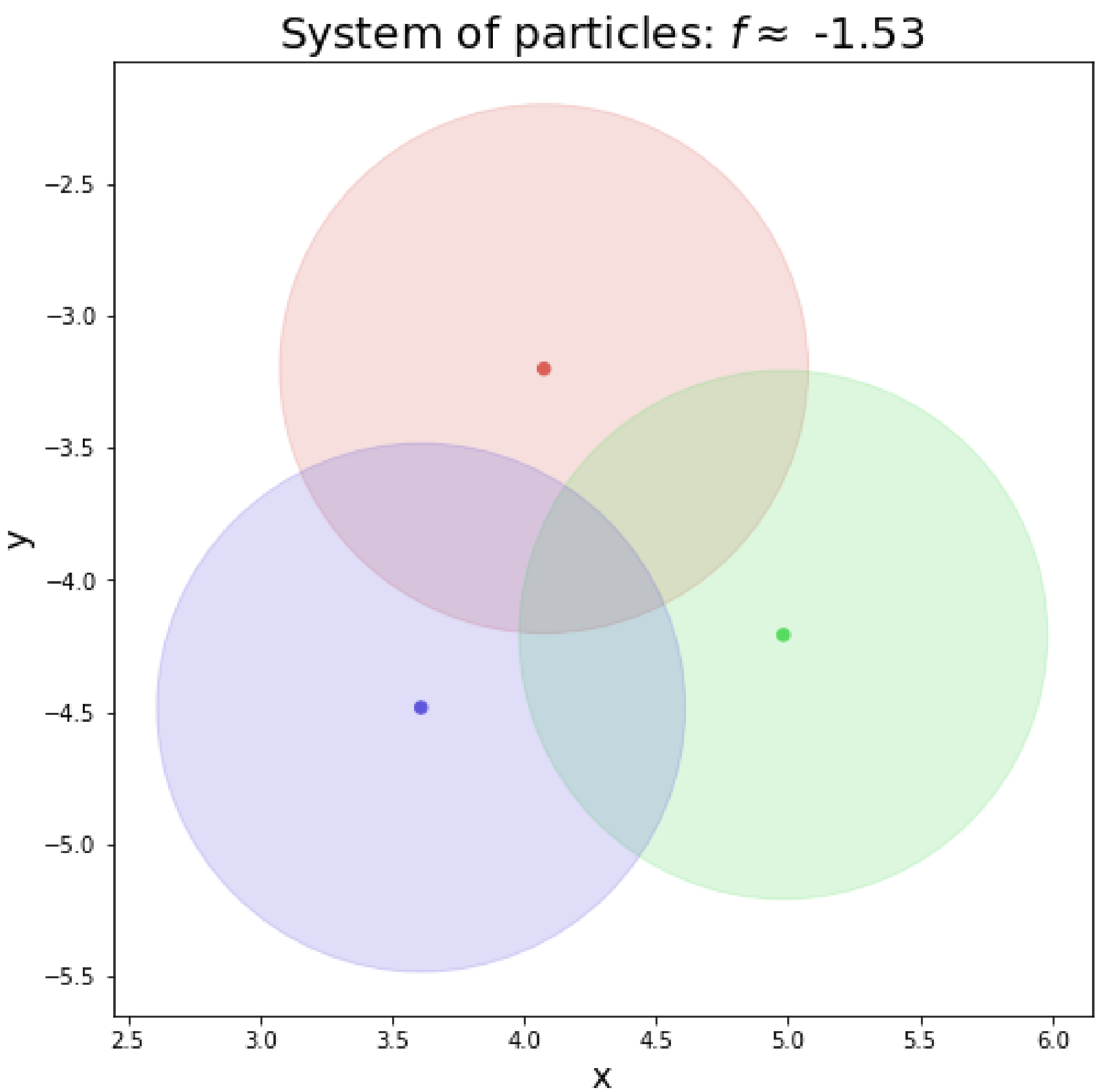


[54]:
```
B0 = np.identity(2*m)

Pn_3_1, P_vec, f_vec_3_1, i = BFGS(P0_3, B0, beta=0.8, maxIter=10000)
```

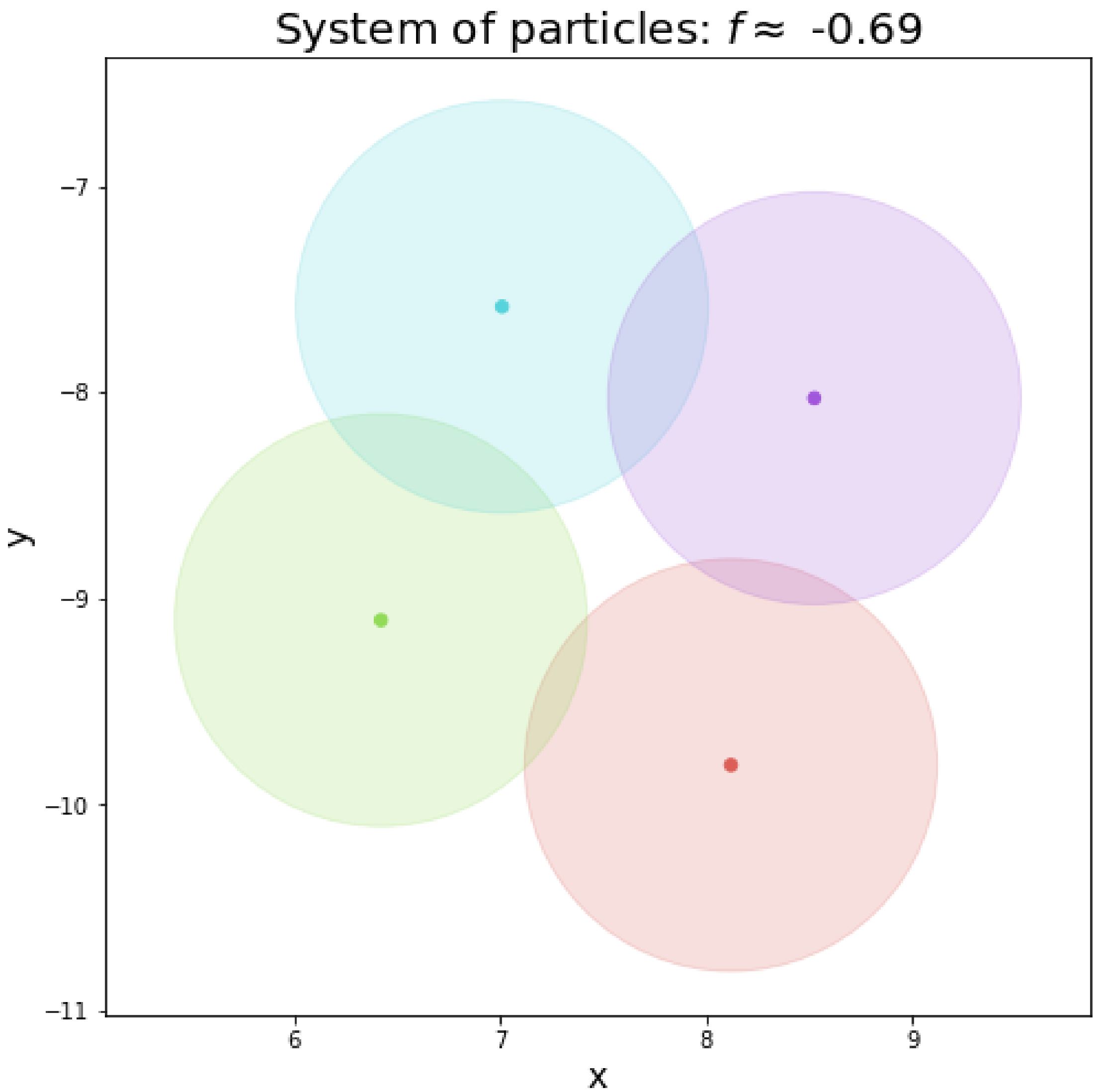
[55]:
```
fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_3_1, f_vec_3_1[-1])
```



[56]:
```
Pn_3_2, P_vec, f_vec_3_2, i = BFGS(P0_3, B0, beta=0.8, maxIter=10000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_3_2, f_vec_3_2[-1])
```

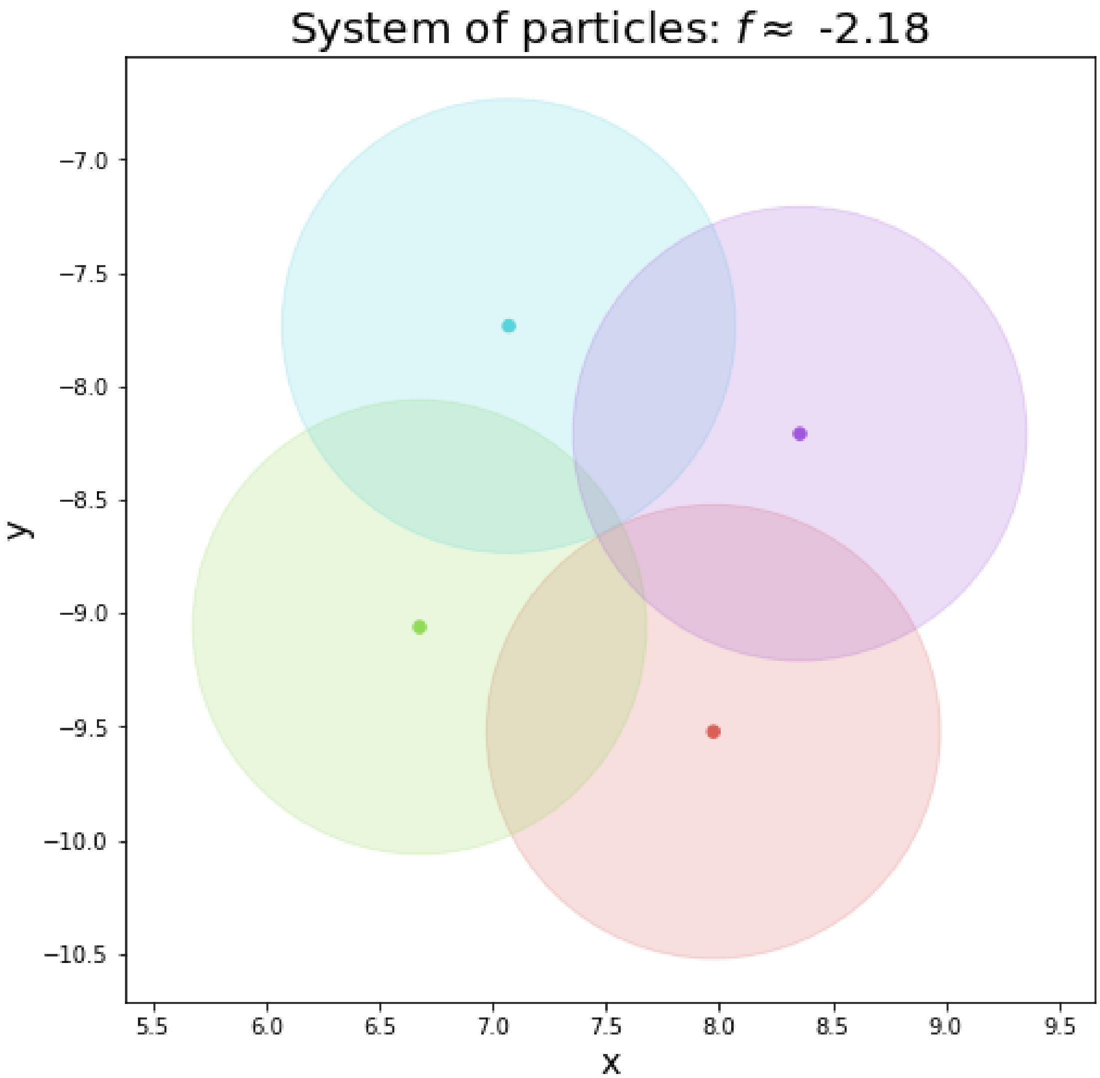


[74]:
```
# Run the algorithm with m = 4
m = 4

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_4 = init_P(m, dist_param=2.5)
plot_particles(ax, P0_4, f(P0_4))
```

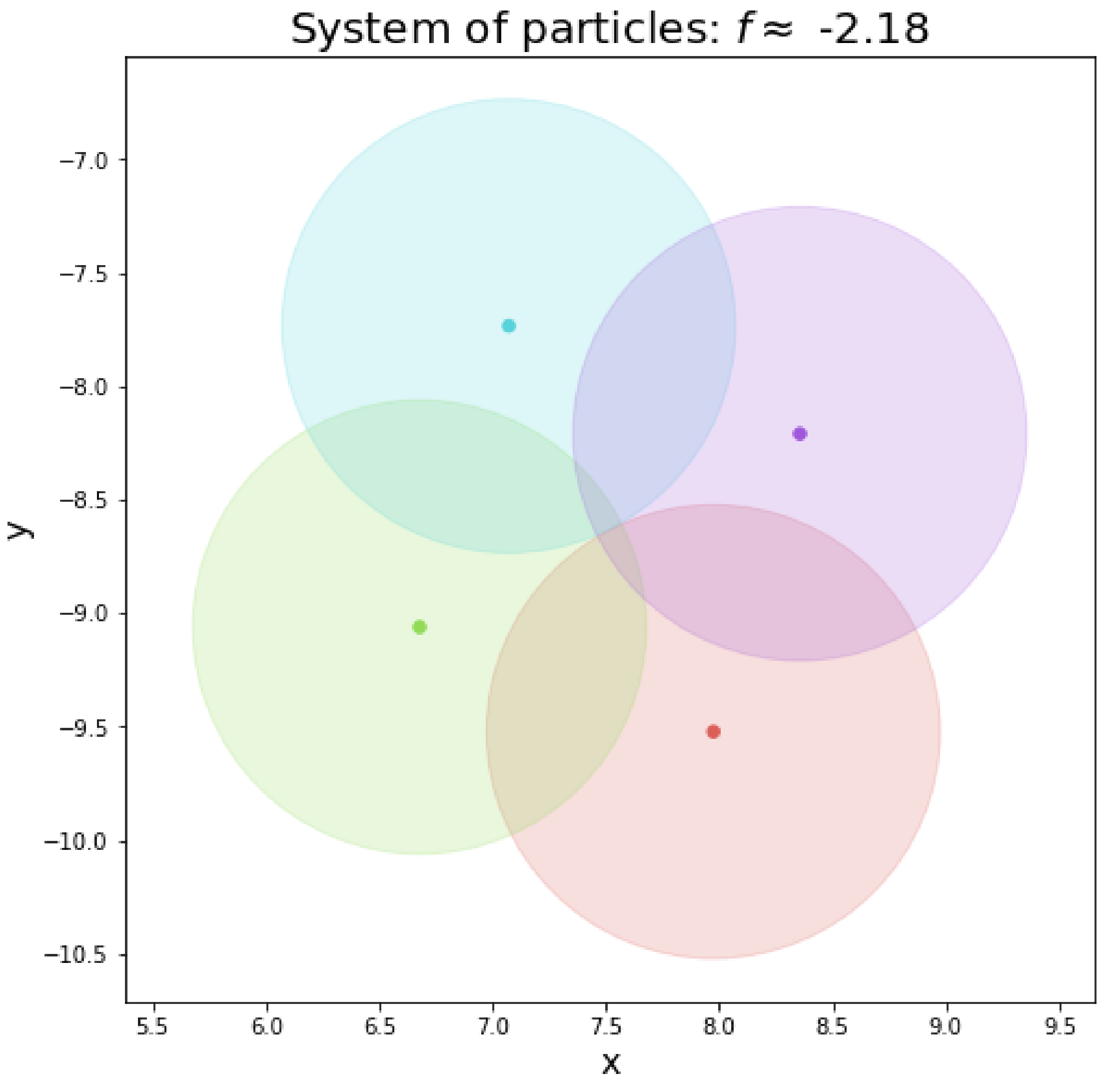


[75]:
```
B0 = np.identity(2*m)

Pn_4_1, P_vec, f_vec_4_1, i = BFGS(P0_4, B0, beta=0.8, maxIter=20000)
```

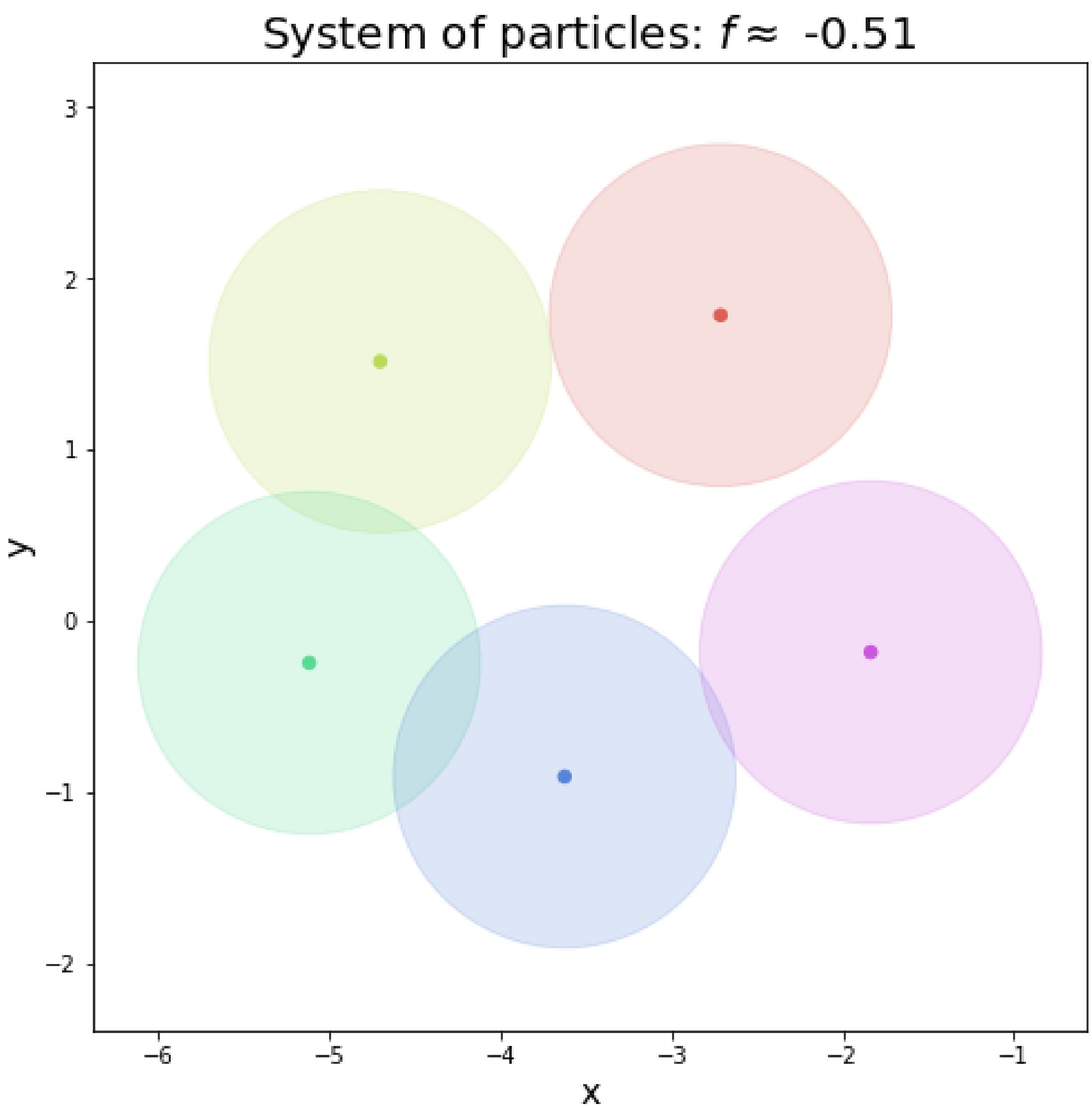
[76]:
```
fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_4_1, f_vec_4_1[-1])
```



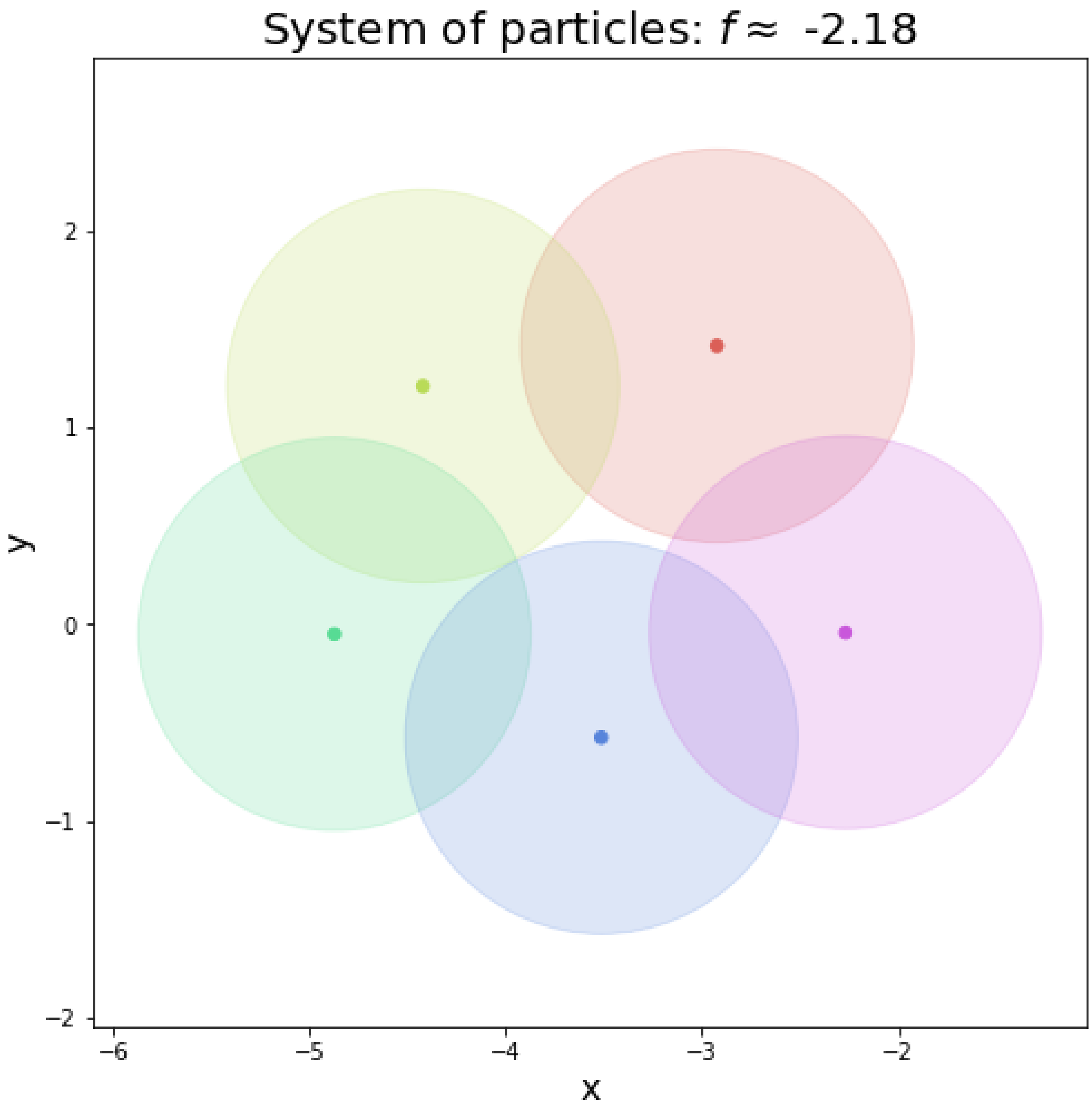
[77]:
```
Pn_4_2, P_vec, f_vec_4_2, i = BFGS(P0_4, B0, beta=0.8, maxIter=20000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_4_2, f_vec_4_2[-1])
```



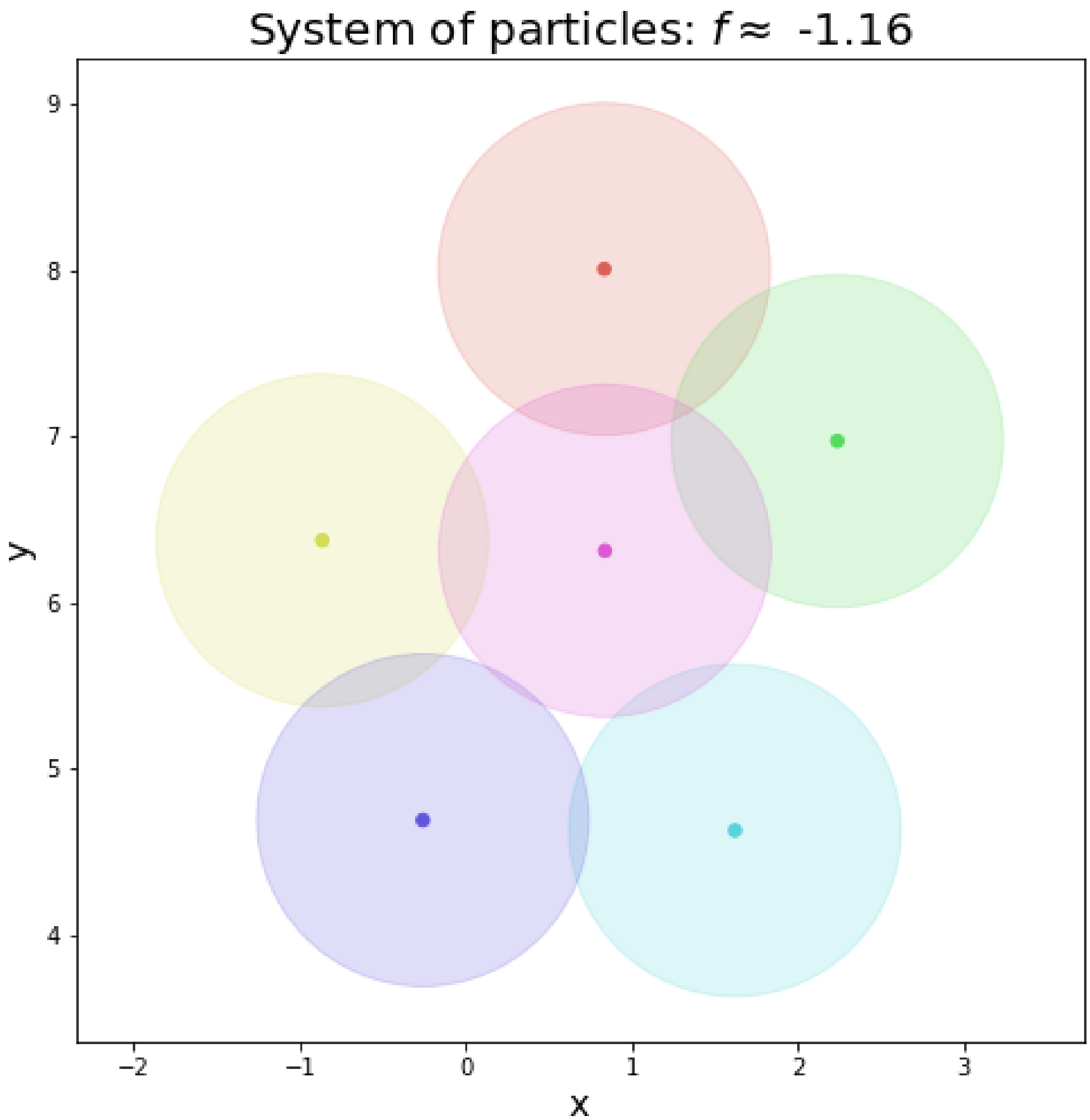
[87]:
```
# Run the algorithm with m = 5
m = 5

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_5 = init_P(m, dist_param=3.5)
plot_particles(ax, P0_5, f(P0_5))
```



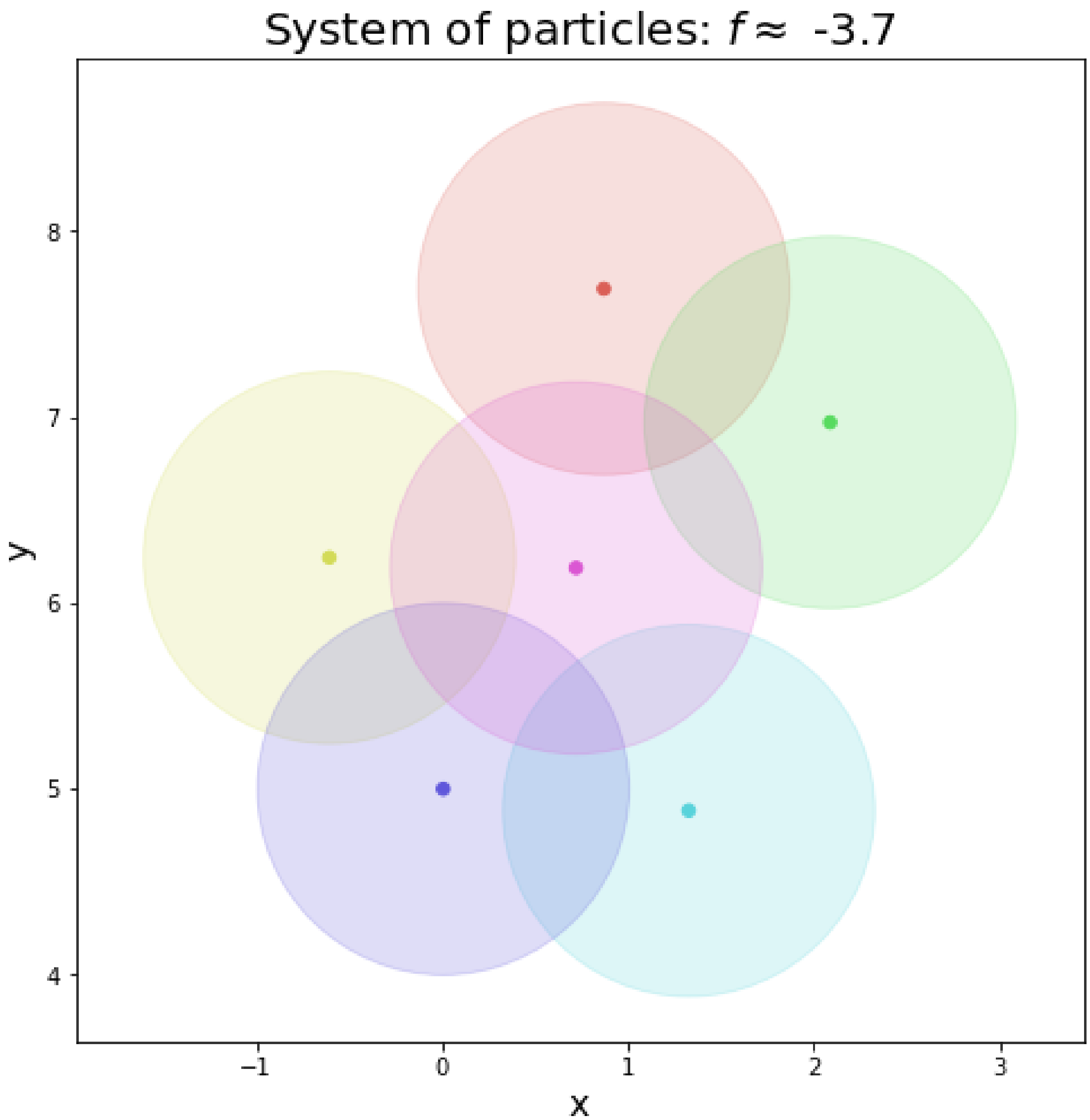
[88]:
```
B0 = np.identity(2*m)

Pn_5_1, P_vec, f_vec_5_1, i = BFGS(P0_5, B0, beta=0.8, maxIter=20000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_5_1, f_vec_5_1[-1])
```



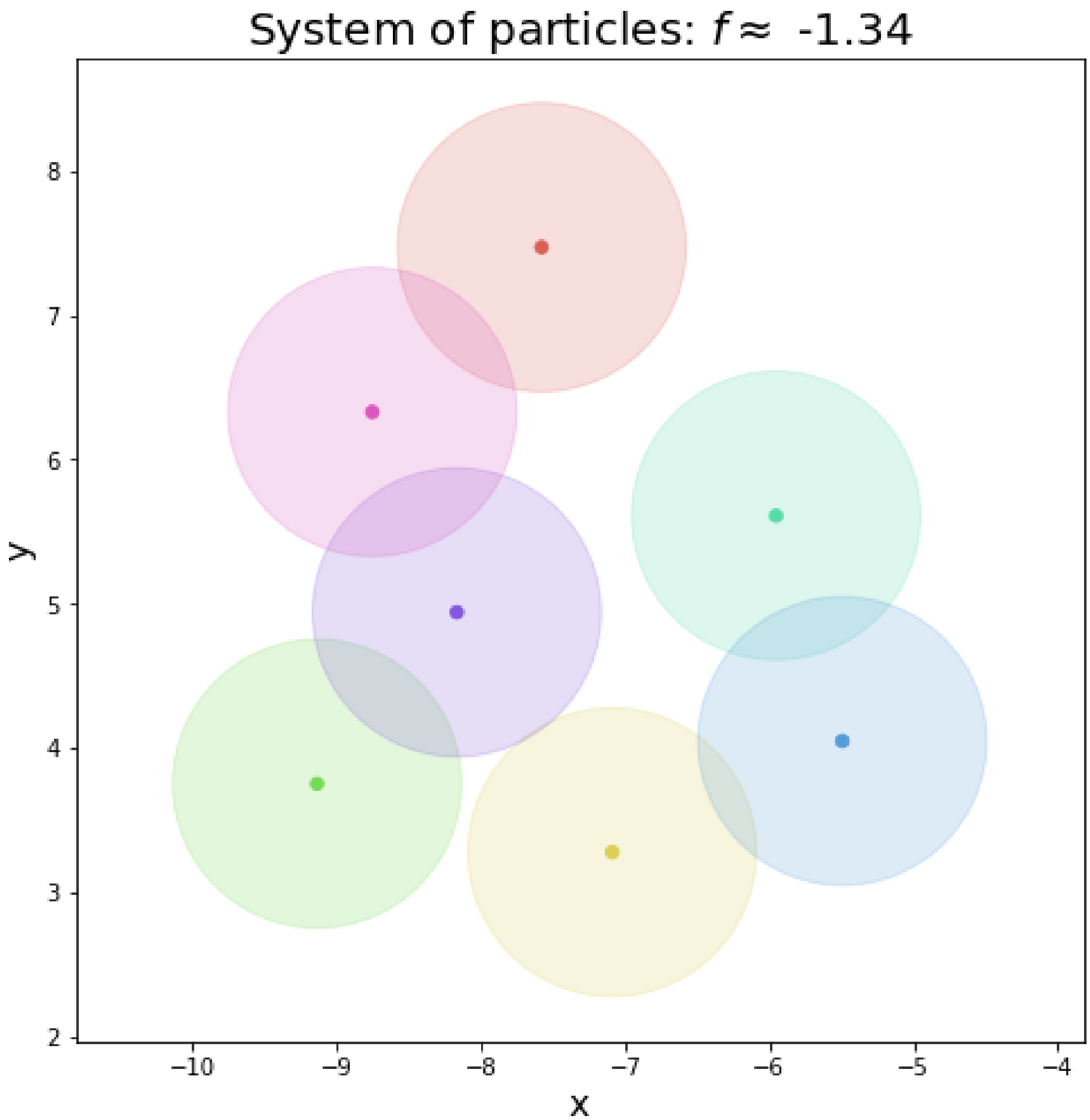
[90]:
```
# Run the algorithm with m = 6
m = 6

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_6 = init_P(m, dist_param=4)
plot_particles(ax, P0_6, f(P0_6))
```



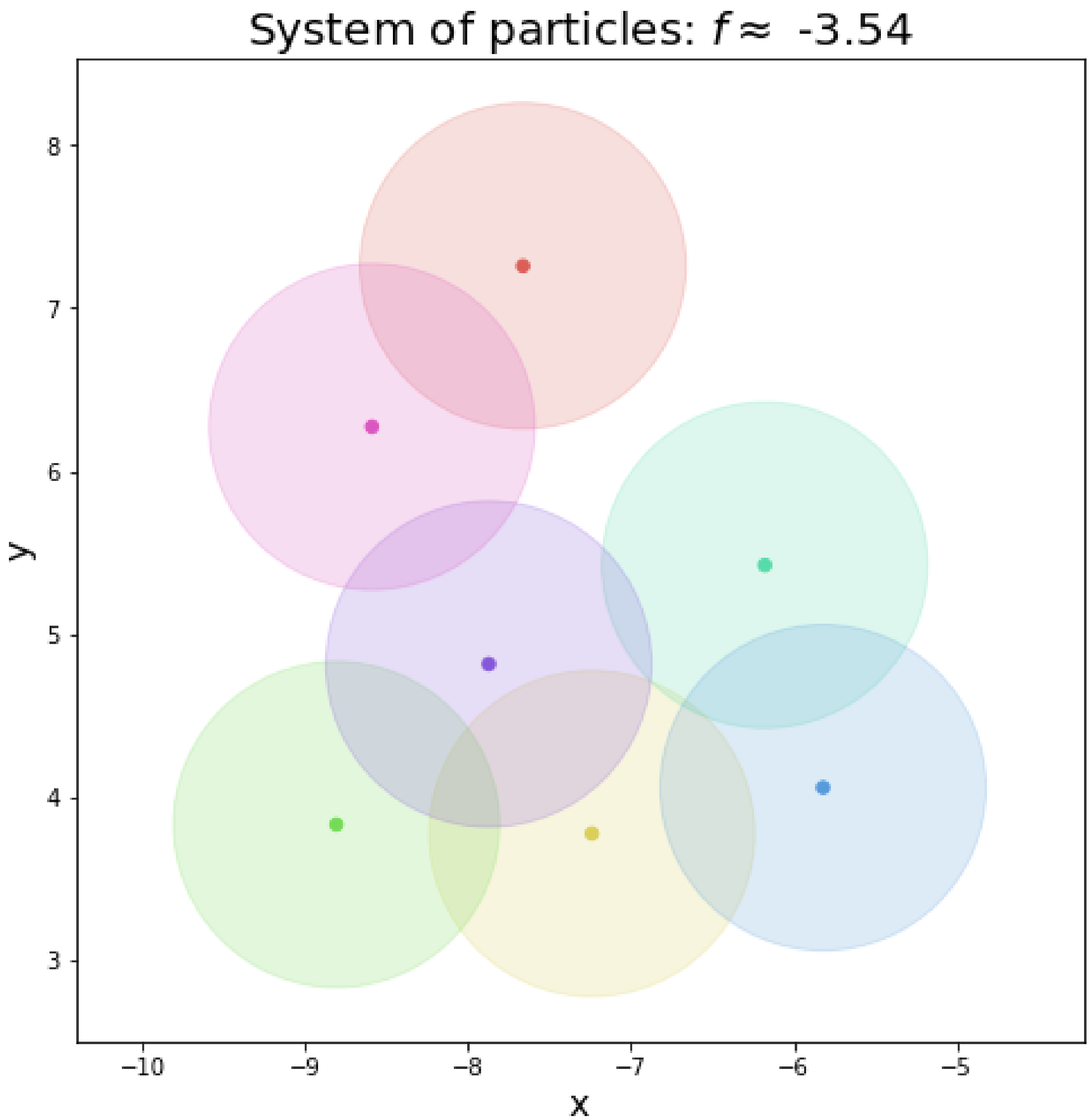
[92]:
```
B0 = np.identity(2*m)

Pn_6_1, P_vec, f_vec_6_1, i = BFGS(P0_6, B0, beta=0.8, maxIter=50000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_6_1, f_vec_6_1[-1])
```



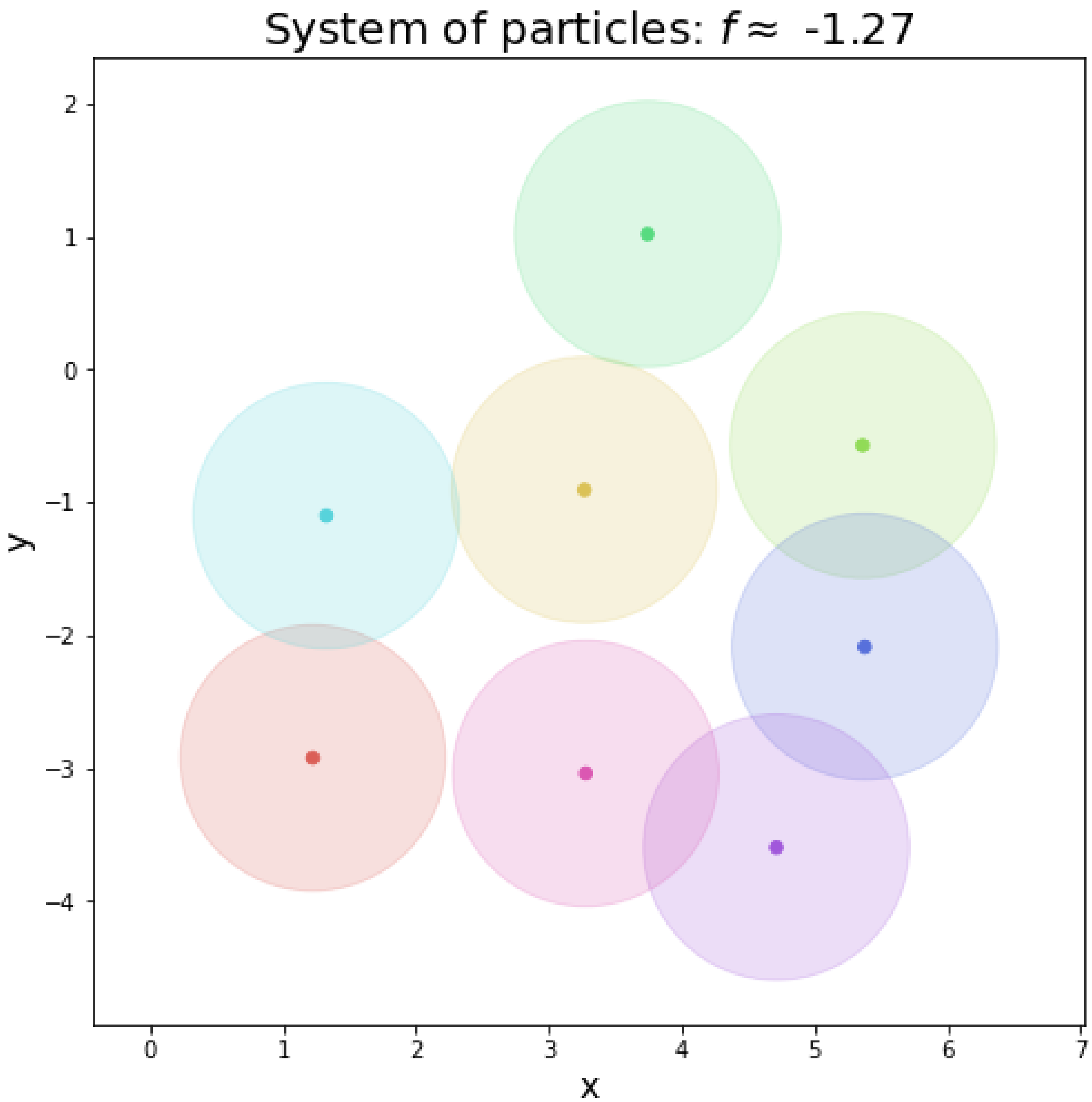
[94]:
```
# Run the algorithm with m = 7
m = 7

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_7 = init_P(m, dist_param=4.5)
plot_particles(ax, P0_7, f(P0_7))
```



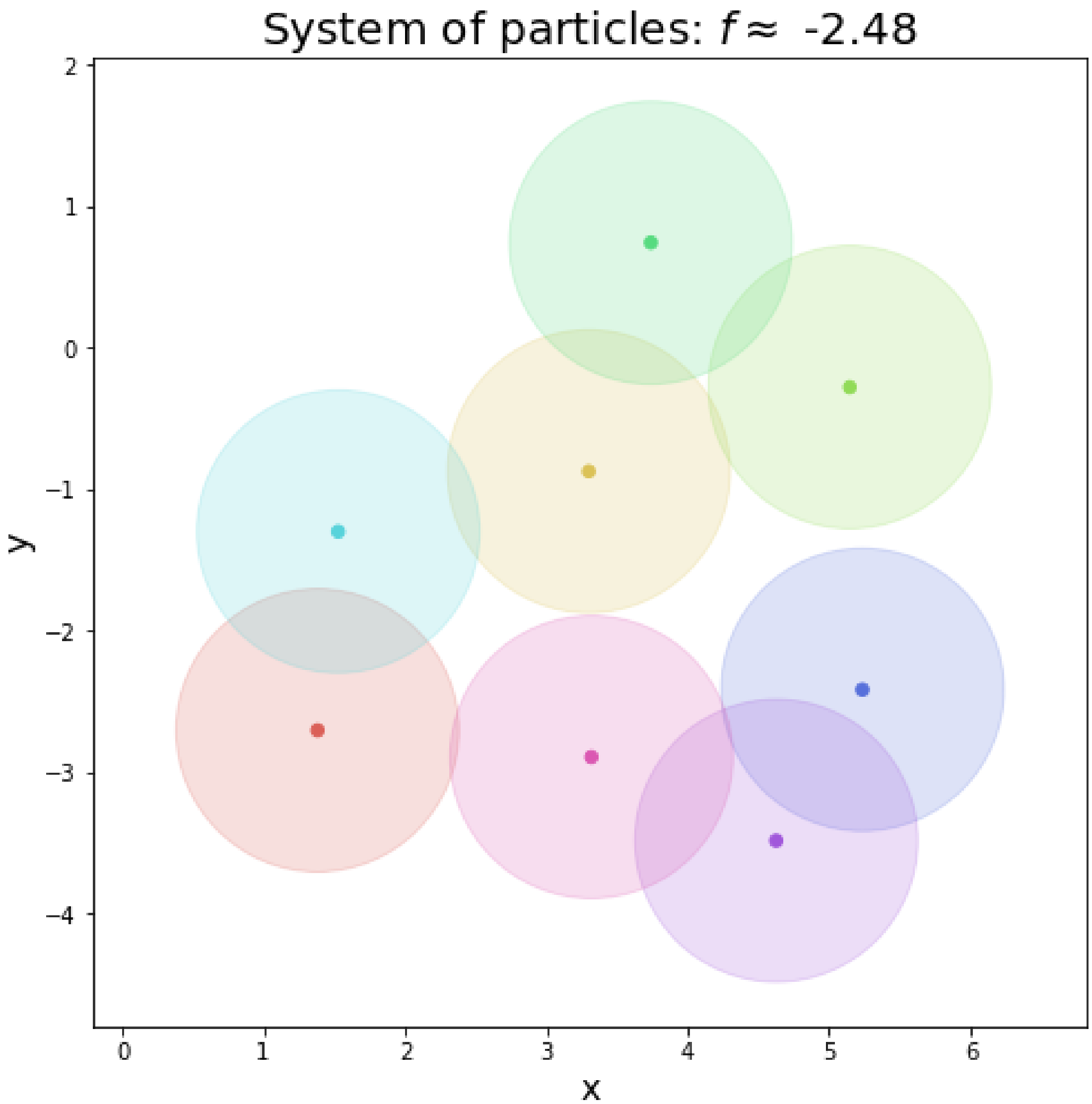
[95]:
```
B0 = np.identity(2*m)

Pn_7_1, P_vec, f_vec_7_1, i = BFGS(P0_7, B0, beta=0.8, maxIter=20000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_7_1, f_vec_7_1[-1])
```



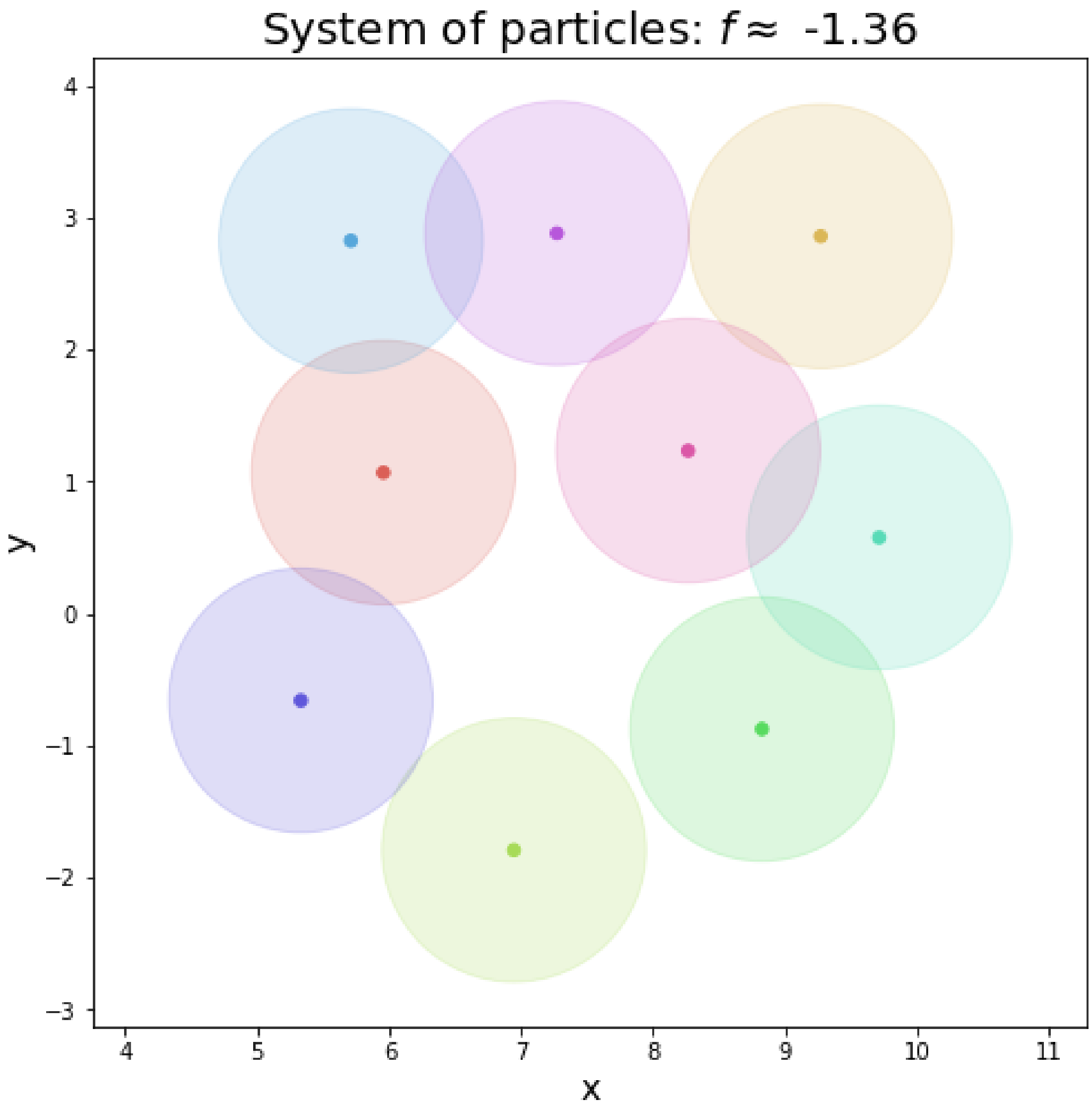
[97]:
```
# Run the algorithm with m = 8
m = 8

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_8 = init_P(m, dist_param=5)
plot_particles(ax, P0_8, f(P0_8))
```



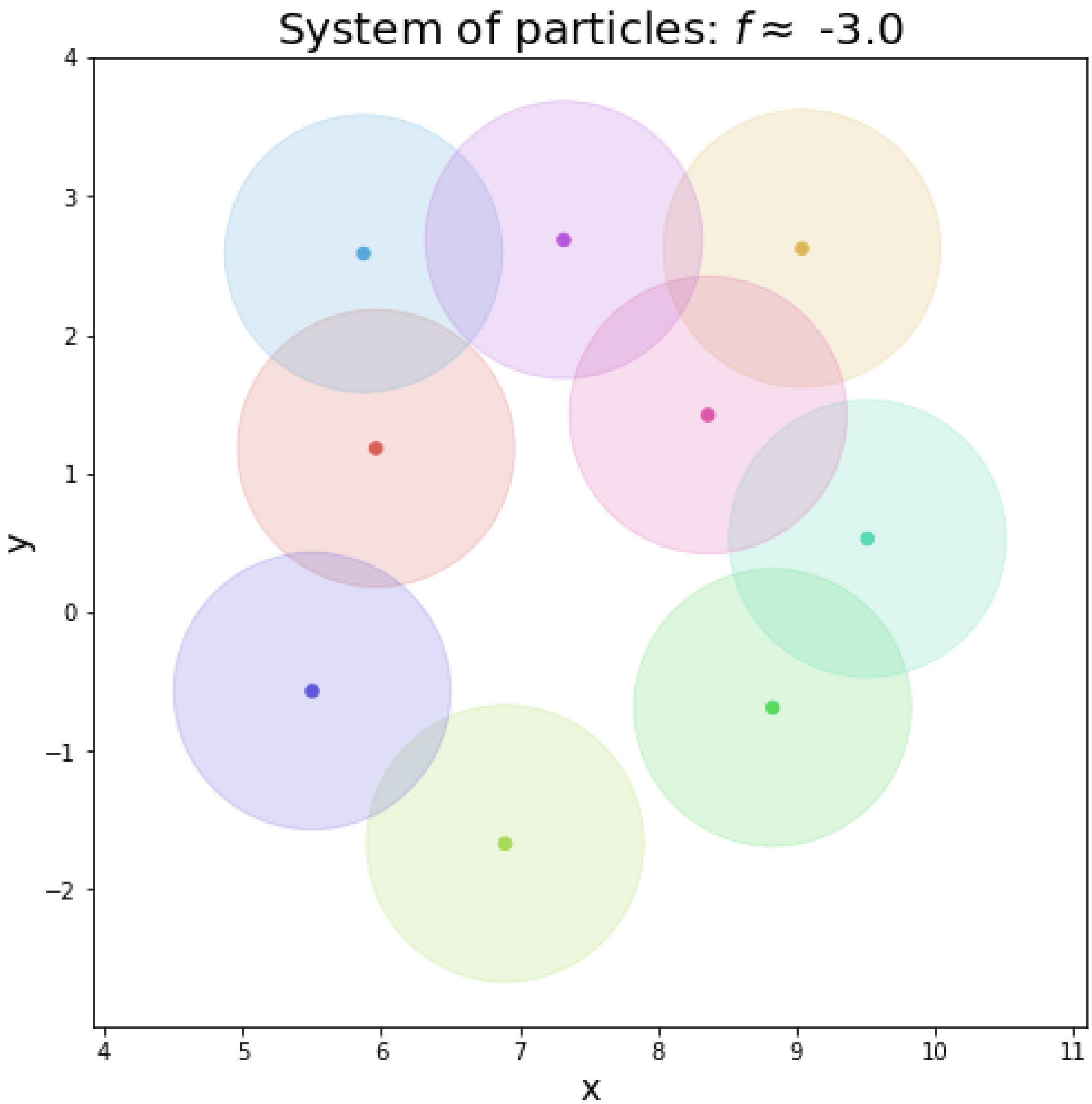
[98]:
```
B0 = np.identity(2*m)

Pn_8_1, P_vec, f_vec_8_1, i = BFGS(P0_8, B0, beta=0.8, maxIter=20000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_8_1, f_vec_8_1[-1])
```



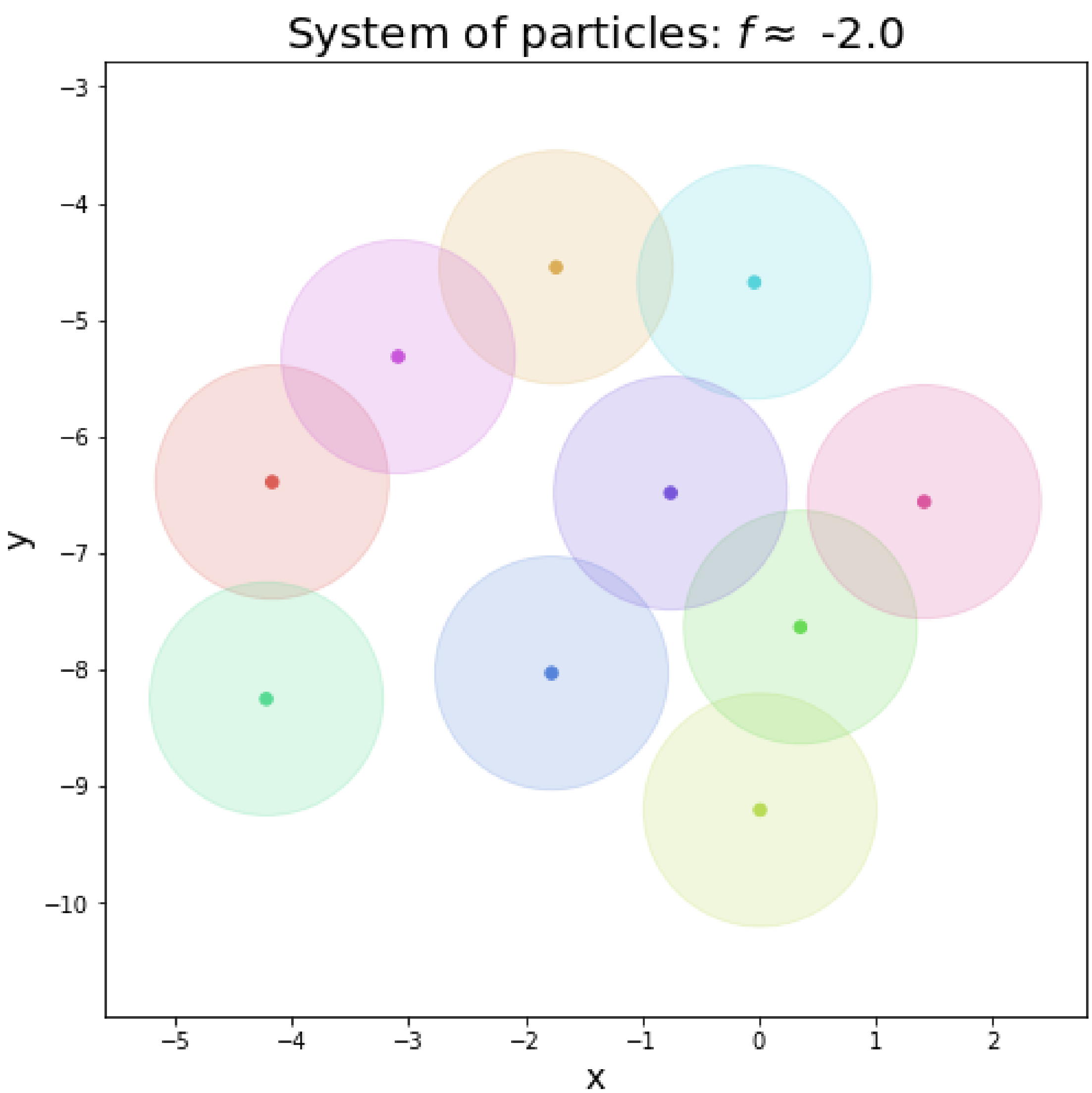
[101]:
```
# Run the algorithm with m = 9
m = 9

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_9 = init_P(m, dist_param=5.5)
plot_particles(ax, P0_9, f(P0_9))
```



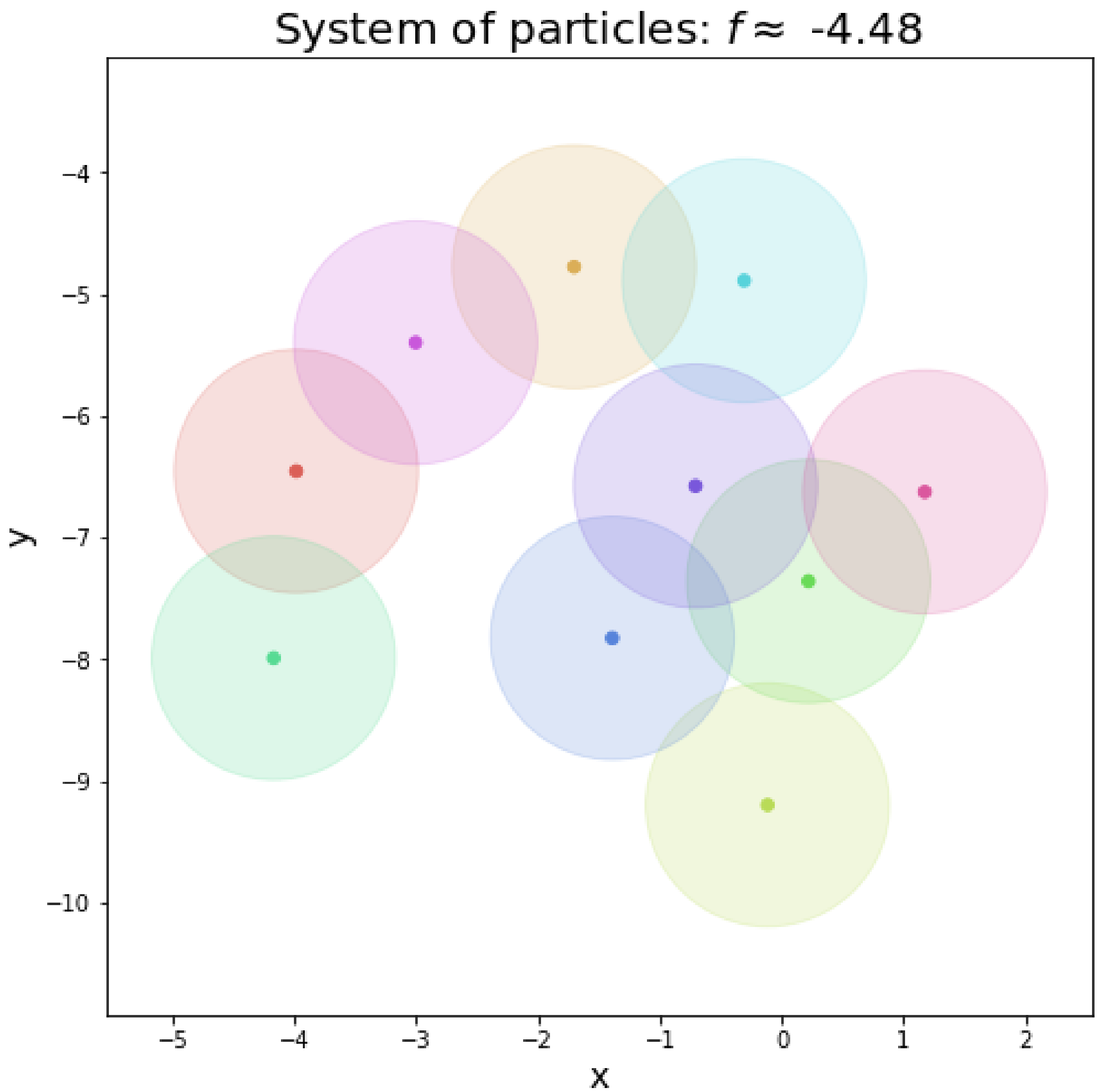
[102]:
```
B0 = np.identity(2*m)

Pn_9_1, P_vec, f_vec_9_1, i = BFGS(P0_9, B0, beta=0.8, maxIter=20000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_9_1, f_vec_9_1[-1])
```



[104]:
```
# Run the algorithm with m = 10
m = 10

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()

P0_10 = init_P(m, dist_param=6)
plot_particles(ax, P0_10, f(P0_10))
```



[105]:
```
B0 = np.identity(2*m)

Pn_10_1, P_vec, f_vec_10_1, i = BFGS(P0_10, B0, beta=0.8, maxIter=20000)

fig = plt.figure(figsize=(8, 8))
ax = plt.gca()
plot_particles(ax, Pn_10_1, f_vec_10_1[-1])
```


System of particles: f ≈ -4.48
y
−4
−5
−6
−7
−8
−9
−10
x
−5
−4
−3
−2
−1
0
1
2

[ ]: