Lab 2: Minimization using Conjugate Gradient / Steepest Descent Method

Shreyansh Agarwal : B23PH1024

Aim :- Given 10 particles ($m=1$) in a box of $10\times10\times10$ units under influence of gravitation potential energy $U(r_{ij}) = \frac{-1}{r_{ij}}$. What is the minimized position of the 10 particles using conjugate gradient and steepest descent method.

Theory:- 1. Steepest Descent Algorithm:- Given $f(x)$ s.t. $x \in \mathbb{R}^n$ and $f(x)$ is differentiable,

$$f(x+d) = f(x) + \nabla f(x)^T d$$

where $d = -\nabla f(x)$.

Algorithm :-

```
I.   Given $x^0$, set $k := 0$
II.  $d^k = -\nabla f(x^k)$. If $d^k = 0$ stop.
III. Solve min $(f(x^k + \alpha d^k))$ fro stepsize $\alpha^k$
IV.  $x^{k+1} \leftarrow x^k + \alpha^k d^k$, $k \leftarrow k+1$. Go to Step II.
```

2. Conjugate Gradient Algorithm:- Given $f(x) = \frac{1}{2}x^T A x - bx$

$$\nabla f(x) = Ax - b$$

Solution of min $f(x)$ is equivalent to $\nabla f(x) = 0 \Rightarrow Ax - b = 0$.

Algorithm:-

```
i. $x^0$, $k := 0$
ii. While $k \leq$ num_Iterations:
iii.        $r_k = b - Ax_k$
iv.         If $r_k <$ threshold return $x_0$
v.          $p_k := r_k$
vi.         $k := 0$
vii.        $\alpha_k = r_k^T r_k / p^{kT} A p^k$
viii.       $x_{k+1} = x_k + \alpha_k^* p_k$
```

Procedure :- i. Initialize 10 particles in a box of length 10×10×10.

ii. Use steepest descent algorithm & CG algorithm to get final positions.

iii. Some points:-

a) Use $U(r_{ij}) = \frac{-1}{r_{ij}+\varepsilon}$ where ε is a small padding used to avoid division by zero. It is equivalent to minimum allowed distance between 2 particles.

b) $f(x) = \sum_{i<j} U(r_{ij}) \longrightarrow$ gives total energy of system, the function to be minimized.

c) $\nabla_i f(x) = \frac{\partial}{\partial x_i} \sum_{j \neq i} \frac{-1}{r_{ij}+\varepsilon} = \sum_{j \neq i} \frac{1}{(r_{ij}+\varepsilon)^2} \frac{\partial}{\partial x_i} (r_{ij} = \|x_i - x_j\|)$

$$= \sum_{j \neq i} \frac{(x_i - x_j)}{(r_{ij}+\varepsilon)^3}$$

where $x_i \in \mathbb{R}^3$, $x_k \in X$ $\quad X \in \mathbb{R}_3^{10}$

IV. Line Search Procedure to get α_k for SD:-

```
a) Initialize $\alpha = 1$, $\beta = 0.5$, $c = 10^{-4}$
b) $E_{curr} = E(x_k)$
c) While $\alpha > 0$:
       $x_{k+1} = x_k + \alpha * d$
       $E_{NEW} = E(x_{k+1})$
       if $E_{NEW} \leq E_{curr} + c * \alpha * Grad . d$
           return $\alpha$
       $\alpha = \alpha * \beta$
```

V. Use non-linear CG:-

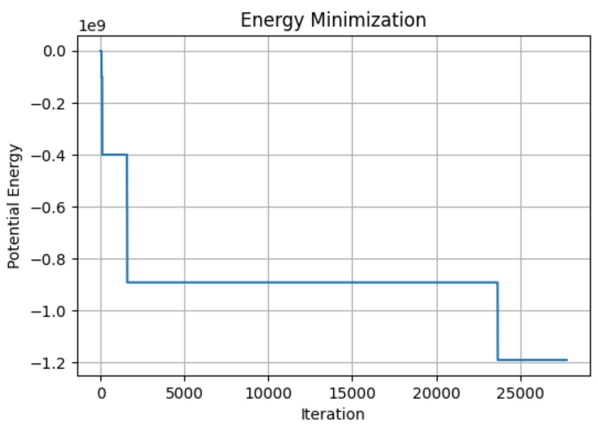
```
a) $g_0 = \nabla f(x_0)$ , $d_0 = -g_0$ ; $k := 0$
b) while $k <= num\_iterations$:
c)     $\alpha_k = \min_{\alpha} f(x_k + \alpha_k d_k)$
d)     $x_{k+1} = x_k + \alpha_k d_k$
e)     $g_{k+1} = \nabla f(x_{k+1})$
f)     $\beta_{k+1} = \frac{\|g_{k+1}\|^2}{\|g_k\|^2}$ → Fletcher Reeves
g)     $d_{k+1} = -g_{k+1} + \beta_{k+1} d_k$
```

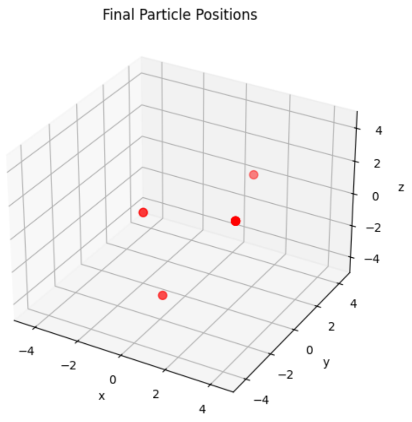
Result:

1. Steepest Descent:

Final Energy = -1191025259.17

X	Y	Z
0.64573321	4.55737053	-0.1424616
4.98886103	-4.98880342	4.98545668
4.98886103	-4.98880342	4.98545668
4.98886103	-4.98880342	4.98545668
0.68656355	-2.94197096	-2.53177161
-4.98188663	4.98884991	-4.98529682
-4.98188663	4.98884991	-4.98529682
-4.98188663	4.98884991	-4.98529682
-4.98188663	4.98884991	-4.98529682
4.98886103	-4.98880342	4.98545668

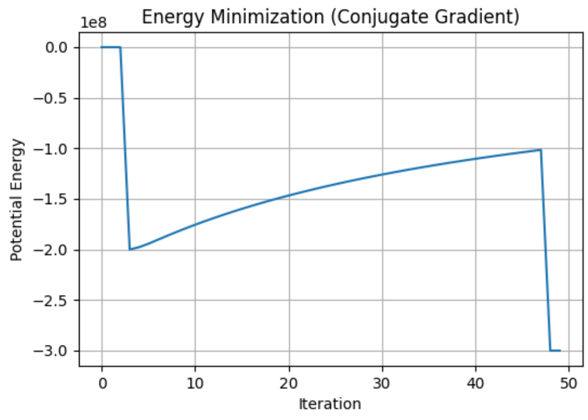


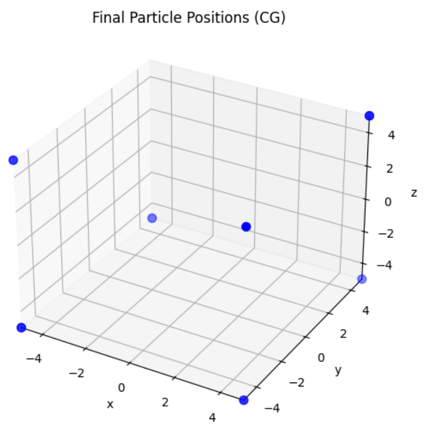


2. Conjugate Gradient:

Final Energy = -300000003.37

X	Y	Z
5	-5	-5
-5	5	-5
5	5	5
-5	-5	-5
-5	-5	-5
5	-5	5
-5	-5	5
-5	5	-5
5	5	5
5	5	-5


Energy Minimization (Conjugate Gradient)
1e8
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
Potential Energy
0
10
20
30
40
50
Iteration


Final Particle Positions (CG)
x
y
z
−4
−2
0
2
4

Code:

Lab2-CG_SD.py

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

class SteepestDescent:
    def __init__(self, particles, n_particles, box_length, tol=1e-12, max_iter=100000, min_particle_dist=1e-8):
        self.n = n_particles
        self.L = box_length
        self.tol = tol
        self.max_iter = max_iter
        self.Eplison = min_particle_dist
        self.Energies = []
        self.positions = particles

    def potential_energy(self, positions=None):
        if positions is None:
            positions = self.positions
        U = 0.0
        for i in range(self.n):
            for j in range(i+1, self.n):
                rij = np.linalg.norm(positions[i] - positions[j])
                U += -1.0 / (rij + self.Eplison)
        return U

    def gradient(self, positions=None):
        if positions is None:
            positions = self.positions
        grad = np.zeros_like(positions)
        for i in range(self.n):
            for j in range(self.n):
                if i != j:
                    rij_vec = positions[i] - positions[j]
                    rij = np.linalg.norm(rij_vec)
                    if rij > 1e-8:
                        grad[i] += rij_vec / ((rij + self.Eplison)**3)
        return grad

    def line_search(self, direction, grad):
        alpha = 1.0
        beta = 0.5
        c = 1e-4

        current_energy = self.potential_energy()
        while alpha > 1e-8:
            new_positions = self.positions + alpha * direction
            new_positions = np.clip(new_positions, -self.L/2, self.L/2)
            new_energy = self.potential_energy(new_positions)

            # Armijo condition
            if new_energy <= current_energy + c * alpha * np.sum(grad * direction):
                return alpha
            alpha *= beta
        return alpha

    def minimize(self):
        prev_energy = self.potential_energy()

        if self.Energies == []:
            self.Energies.append(prev_energy)

        for step in range(self.max_iter):
            grad = self.gradient()
            direction = -grad

            if np.linalg.norm(grad)<=self.tol:
                print(f"Converged at step {step}, Energy = {self.Energies[-1]:.6f}")
                break

            alpha = self.line_search(direction, grad)
            self.positions += alpha * direction
            self.positions = np.clip(self.positions, -self.L/2, self.L/2)

            energy = self.potential_energy()
            self.Energies.append(energy)
            prev_energy = energy

        return self.positions, self.Energies[-1]

    def energy_plot(self):
        plt.figure(figsize=(6,4))
        plt.plot(self.Energies)
        plt.xlabel("Iteration")
        plt.ylabel("Potential Energy")
        plt.title("Energy Minimization")
        plt.grid(True)
        plt.show()

    def final_positions_plot(self):
        fig = plt.figure(figsize=(6,6))
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(self.positions[:,0], self.positions[:,1], self.positions[:,2], c='red',
s=50)
        ax.set_xlim([-self.L/2, self.L/2])
        ax.set_ylim([-self.L/2, self.L/2])
        ax.set_zlim([-self.L/2, self.L/2])
        ax.set_title("Final Particle Positions")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_zlabel("z")
        plt.show()

class ConjugateGradient:
    def __init__(self, particles, n_particles, box_length, tol=1e-12, max_iter=100000,
min_particle_dist=1e-8):
        self.n = n_particles
        self.L = box_length
        self.tol = tol
        self.max_iter = max_iter
        self.Eplison = min_particle_dist
        self.Energies = []
        self.positions = particles

    def potential_energy(self, positions=None):
        if positions is None:
            positions = self.positions
        U = 0.0
        for i in range(self.n):
            for j in range(i+1, self.n):
                rij = np.linalg.norm(positions[i] - positions[j])
                U += -1.0 / (rij + self.Eplison)
        return U

    def gradient(self, positions=None):
        if positions is None:
            positions = self.positions
        grad = np.zeros_like(positions)
        for i in range(self.n):
            for j in range(self.n):
                if i != j:
                    rij_vec = positions[i] - positions[j]
                    rij = np.linalg.norm(rij_vec)
                    if rij > 1e-8:
                        grad[i] += rij_vec / ((rij + self.Eplison)**3)
        return grad

    def line_search(self, direction, grad):
        alpha = 1.0
        beta = 0.5
        c = 1e-4

        current_energy = self.potential_energy()
        while alpha > 1e-8:
            new_positions = self.positions + alpha * direction
            new_positions = np.clip(new_positions, -self.L/2, self.L/2)
            new_energy = self.potential_energy(new_positions)

            # Armijo condition
            if new_energy <= current_energy + c * alpha * np.sum(grad * direction):
                return alpha
            alpha *= beta
        return alpha

    def minimize(self):
        grad = self.gradient()
        direction = -grad
        prev_energy = self.potential_energy()
        self.Energies.append(prev_energy)

        for step in range(self.max_iter):
            if np.linalg.norm(grad)<=self.tol:
                print(f"Converged at step {step}, Energy = {self.Energies[-1]:.6f}")
                break

            alpha = self.line_search(direction, grad)
            self.positions += alpha * direction
            self.positions = np.clip(self.positions, -self.L/2, self.L/2)

            new_grad = self.gradient()
            energy = self.potential_energy()
            self.Energies.append(energy)

            # Fletcher-Reeves beta update
            beta = np.sum(new_grad*new_grad) / (np.sum(grad*grad) + 1e-12)
            direction = -new_grad + beta * direction

            grad = new_grad
            prev_energy = energy

        return self.positions, self.Energies[-1]

    def energy_plot(self):
        plt.figure(figsize=(6,4))
        plt.plot(self.Energies)
        plt.xlabel("Iteration")
        plt.ylabel("Potential Energy")
        plt.title("Energy Minimization (Conjugate Gradient)")
        plt.grid(True)
        plt.show()

    def final_positions_plot(self):
        fig = plt.figure(figsize=(6,6))
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(self.positions[:,0], self.positions[:,1], self.positions[:,2], c='blue', s=50)
        ax.set_xlim([-self.L/2, self.L/2])
        ax.set_ylim([-self.L/2, self.L/2])
        ax.set_zlim([-self.L/2, self.L/2])
        ax.set_title("Final Particle Positions (CG)")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_zlabel("z")
        plt.show()


# Example usage:
if __name__ == "__main__":
    particles = np.random.uniform(-5, 5, (10,3))

    sd = SteepestDescent(particles, n_particles=10, box_length=10)
    final_positions, final_energy = sd.minimize()
    print("Final positions SD:\n", final_positions)
    print("Final energy:", final_energy)
    sd.energy_plot()
    sd.final_positions_plot()

    cg = ConjugateGradient(particles, n_particles=10, box_length=10)
    final_positions, final_energy = cg.minimize()
    print("Final positions CG:\n", final_positions)
    print("Final energy:", final_energy)
    cg.energy_plot()
    cg.final_positions_plot()

```