# Numerical analysis: P2

Students: Bodong Liu (bl576), Ryan Wu (rw645)
Date: May 3, 2025

---

## Outline

1. **Introduction**
2. **Optimization Methods**
   (a) Gradient Descent
   (b) Quasi-Newton (BFGS)
3. **Small System Results**
   (a) $N = 2$
   (b) $N = 3$
4. **Convergence Analysis**
5. **Discussion**
6. **References**

# 1 Introduction

In this project, we are trying to solve the problem of finding lowest-energy configurations of $N$ particles interacting according to a simplified version of Lennard–Jones potential in 3D space. We fix one particle to be translational invariant, optimizing the remaining $3(N-1)$ coordinates:

$$\min_{x_2,\dots,x_N} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} V_{ij},$$

where $r_{ij} = \|x_i - x_j\|$ and

$$V_{ij}(r) = r^{-12} - 2\,r^{-6}.$$

Our goals are:

- Implement two solver methods—Gradient Descent and a Quasi-Newton BFGS—each with a backtracking line search and sensible stopping criteria.
- Validate these solvers by locating global minima for small systems ($N = 2, 3$) and argue uniqueness of the solutions.
- Analyze convergence behavior.
- Explore further how the solvers behave with greater $N$.

# 2 Optimization Methods

## 2.1 Gradient Descent

We implement the basic gradient descent method to minimize the Total Lennard–Jones Energy:

$$E(X) = \sum_{1\le i<j\le N} \Big( \|x_i - x_j\|^{-12} - 2\,\|x_i - x_j\|^{-6} \Big),$$

working in the reduced $3(N-1)$-dimensional subspace by fixing $x_1 = (0,0,0)$. Below is a walk-through of the core algorithm. Refer to Algorithm 1 in the "Algorithm" section for a more detailed pseudo-code.

**At each iteration $k$:**

1. **Compute gradient.**
$$g^{(k)} = \nabla E\big(X^{(k)}\big).$$
2. **Line search (Armijo backtracking).** Starting from $\alpha = \alpha_0$, repeatedly shrink $\alpha \leftarrow \beta\,\alpha$ until
$$E\big(X^{(k)} - \alpha\, g^{(k)}\big) \;\le\; E\big(X^{(k)}\big) \;-\; c\,\alpha\,\|g^{(k)}\|^2,$$
with default parameters $\alpha_0 = 1.0$, $\beta = 0.5$, and $c = 10^{-4}$.
3. **Update.**
$$X^{(k+1)} = X^{(k)} - \alpha\, g^{(k)}.$$
4. **Stopping criteria.**
   - Gradient-norm test: $\|g^{(k)}\| \le \epsilon_g$ (e.g. $\epsilon_g = 10^{-8}$).
   - Maximum iterations: $k \ge K_{\max}$ (e.g. $K_{\max} = 2000$).
   - Energy-change test: $\big|E(X^{(k+1)}) - E(X^{(k)})\big| \le \epsilon_E$.

## 2.2 Quasi-Newton (BFGS)

We implement the BFGS method in the reduced subspace of dimension $3(N-1)$ by pinning one atom. Let $x^{(k)}$ and $g^{(k)}$ denote the free coordinates and gradient, and $H_k^{-1}$ the inverse-Hessian approximation. Below is a general walkthrough for each iteration. Refer to Algorithm 2 in the "Algorithm" section for a more detailed pseudo-code.

**At each iteration $k$:**

1. **Compute full gradient.**

$$G^{(k)} = \nabla E\big(X^{(k)}\big), \quad g^{(k)} = \mathrm{vec}\big(G^{(k)}\big)_{\mathrm{free}}.$$

2. **Form search direction.**

$$s^{(k)} = -\,H_k^{-1}\, g^{(k)},$$

where $H_k^{-1} \in \mathbb{R}^{3(N-1)\times 3(N-1)}$ is our current inverse-Hessian approximation.

3. **Line search (Strong Wolfe).** Find the step length $\alpha$ along the full-space direction $P^{(k)}$ by enforcing

$$E\big(X^{(k)}+\alpha\, P^{(k)}\big) \le E\big(X^{(k)}\big)+c_1\,\alpha\,(g^{(k)})^\top s^{(k)} \quad \text{and} \quad \big|\nabla E(X^{(k)}+\alpha\, P^{(k)})^\top P^{(k)}\big| \le -c_2\,(g^{(k)})^\top s^{(k)},$$

with default parameters $c_1 = 10^{-4}$, $c_2 = 0.9$. We use a bracket-and-zoom routine to satisfy the Strong Wolfe conditions (Ghosh, 2021; Nocedal & Wright, 2006).

4. **Update iterate.**

$$x^{(k+1)} = x^{(k)} + \alpha_k\, s^{(k)}, \quad X^{(k+1)} = \mathrm{assemble}(x^{(k+1)}), \quad X^{(k+1)}.\mathrm{requires\_grad\_}() = \texttt{True}.$$

5. **Compute new gradient.**

$$G^{(k+1)} = \nabla E\big(X^{(k+1)}\big), \quad g^{(k+1)} = \mathrm{vec}\big(G^{(k+1)}\big)_{\mathrm{free}}.$$

6. **BFGS update.** Form

$$w^{(k)} = \alpha_k s^{(k)}, \quad y^{(k)} = g^{(k+1)} - g^{(k)}, \quad \rho^{(k)} = \frac{1}{(y^{(k)})^\top w^{(k)}}.$$

- If $(y^{(k)})^\top w^{(k)} < 10^{-8}$, skip update (to avoid a non-positive or numerically unstable $\rho$).
- Otherwise,

$$V = I - \rho^{(k)}\, w^{(k)}(y^{(k)})^\top, \quad H_{k+1}^{-1} = V\, H_k^{-1}\, V^\top + \rho^{(k)}\, w^{(k)}(w^{(k)})^\top.$$

7. **Stopping criteria.**

- Energy-change test: $\big|E(X^{(k+1)}) - E(X^{(k)})\big| \le \epsilon_E$.
- Gradient-norm test: $\|g^{(k+1)}\| \le \epsilon_g$.
- Maximum iterations: $k \ge K_{\max}$.

## 3 Small Systems Results

In this section we apply our Gradient Descent and BFGS implementations to the cases $N = 2$ and $N = 3$. We report the found configurations, energies, and argue for global optimality and uniqueness.

### 3.1 $N = 2$

**Expected solution:** With only two atoms and the simplified L-J potential

$$V(r) = r^{-12} - 2\,r^{-6},$$

we can hand-calculate and obtain the unique minimizer:

$$r^* = 1, \qquad V(r^*) = -1.$$

**Computed result:** Our GD solver converges to

$$X = \begin{bmatrix} (0, 0, 0) \\ (0.7923309,\ -0.3080589,\ 0.5266037) \end{bmatrix}, \quad r_{12} \approx 1, \quad E(X) \approx -1.0000.$$

Our BFGS solver converges to

$$X = \begin{bmatrix} (0, 0, 0) \\ (-0.7923317,\ 0.3080592,\ -0.5266043) \end{bmatrix}, \quad r_{12} \approx 1, \quad E(X) \approx -1.0000.$$

Two solvers produce identical geometry as below.

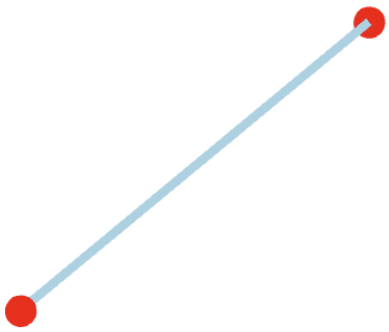


Figure 1: Configuration for $N = 2$.

**Global optimality and uniqueness:** Since there is only one pair interaction, the derivative of potential energy has a single critical point at $r = 1$, which is a global minimum. Hence, this minimization is unique (not counting rotation or translation).

### 3.2 $N = 3$

**Expected solution:** For three atoms, symmetry suggests an equilateral triangle of side length $r^* = 1$. The total energy is

$$E_3 = 3\,V(1) = -3.$$

**Computed result:** Our GD solver converges to:,

$$X \approx \begin{bmatrix} (0,0,0) \\ (-0.8421267,\ 0.3627993,\ -0.3989978) \\ (-0.4070932,\ -0.444385,\ -0.7979956) \end{bmatrix}, \quad r_{ij} \approx 1 \quad \forall 1 \le i < j \le 3, \quad E(X) \approx -3.0000.$$

Our BFGS solver converges to:,

$$X \approx \begin{bmatrix} (0,0,0) \\ (-0.8421709,\ 0.3649025,\ -0.3969839) \\ (-0.4092407,\ -0.4424151,\ -0.7979930) \end{bmatrix}, \quad r_{ij} \approx 1 \quad \forall 1 \le i < j \le 3, \quad E(X) \approx -3.0000.$$

Two solvers produce identical geometry. As we can see from below that the configuration is a equilateral triangle which matches the theoretical solution.

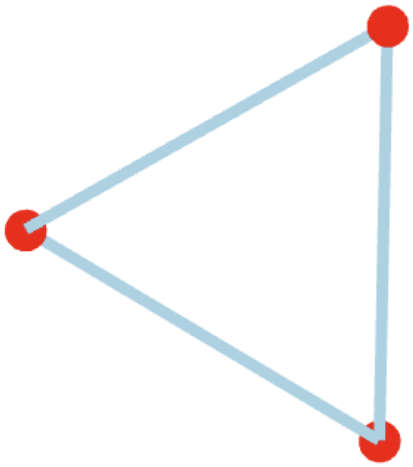


Figure 2: Configuration for $N = 3$

**Global optimality and uniqueness.** By symmetry and convexity of the sum of identical pairwise interactions, the equilateral triangle is the configuration that minimizes $\sum_{i<j} V(\|x_i - x_j\|)$. There are no other forms of triangles achieving $E = -3$. Hence, this minimization is unique (not counting rotation or translation).

# 4 Convergence Analysis

## 4.1 Gradient Descent

In a three atom configuration, we found that globally optimal configurations for N=3 for the gradient descent implementation using line search converges roughly linearly, and in a low numnber of steps. The follows shows the estimated convergence order, where $error_n$ is the absolute difference of energy between the $n-1$ and $n$ iterations.

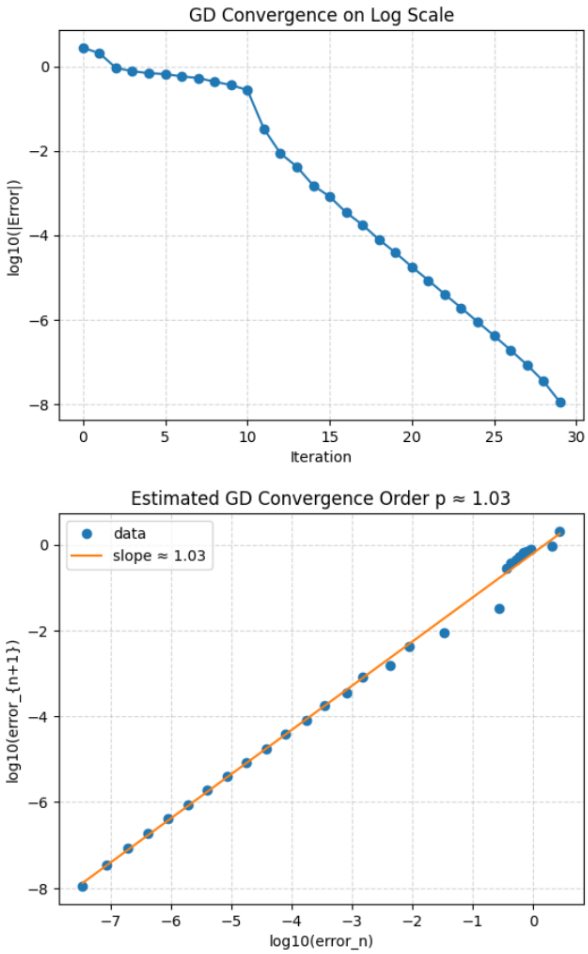



Figure 3: Convergence for Gradient Descent, N=3

These fall in line with our expectations for the performance of Gradient Descent, where it typically has a linear convergence rate (our experimental convergence rate is $\rho = 1.03$).

## 4.2 BFGS

The theoretical order of convergence of BFGS is Q-superlinear, where it should converge faster than linear in specific conditions. However, the conditions for this case are rather strong: the Hessian must be positive definite in the neighborhood of the minimzer, the line searcg must satisfy the strong Wolfe conditions, and the problem is non-degenerate.

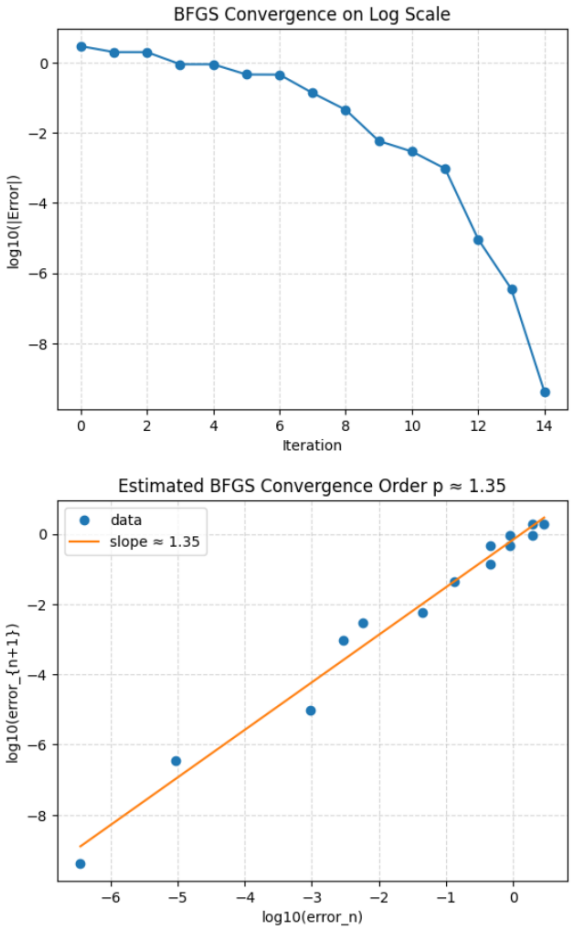



Figure 4: Convergence for BFGS, N=3

Here we see that the there is a superlinear convergence rate (faster than linear, but not quite at quadratic) as expected: $\rho > 1$ and $\rho < 2$. This is the performance that is expected by BFGS, when we enforce the Strong Wolfe Conditions (Armijo + Curvature).

# 5 Discussion

We ran both solvers for greater values of $N$ up to $N = 10$ as reported in Table 1. We also included the simulation results from The Cambridge Cluster Database in the last column to have a better look at how good our solvers are. The Cambridge Cluster Database uses a 'basin-hopping' algorithm for $N < 110$, which are unbiased global optimization methods (The Cambridge Cluster Database). We can see that our solvers are generally doing great. The biggest discrepancy occurs at $N = 10$ with an error of 5.8%. Please refer to page 32 for a visualization of the configurations.

| $N$ | $E$(GD) | $E$(BFGS) | $E$(Cambridge) |
|---|---|---|---|
| 4 | -6.000000 | -6.000000 | -6.000000 |
| 5 | -9.103852 | -9.103852 | -9.103852 |
| 6 | -12.712062 | -12.302927 | -12.712062 |
| 7 | -16.505384 | -16.505384 | -16.505384 |
| 8 | -19.765298 | -19.765297 | -19.821489 |
| 9 | -23.269812 | -23.269812 | -24.113360 |
| 10 | -26.771677 | -26.771677 | -28.422532 |

Table 1: Results from Gradient Descent, BFGS, and The Cambridge Cluster Database

In our initial version of BFGS algorithm, we observed the total energy and gradient norm become NaN after a few iterations. It was because

$$y^{(k)\top} s^{(k)} = \left(g^{(k+1)} - g^{(k)}\right)^{\top} s^{(k)}$$

could become extremely small or even slightly negative when the gradient does not change much plus the numerical noise and inexact line searches. Since the BFGS update uses $\rho = 1/(y^T s)$, a tiny or non-positive denominator sends $\rho$ to huge or undefined values, which in turn causes the inverse Hessian to be not positive definiteness and leads to NaN steps. To fix this, we insert the safeguard

$$\text{if } y^{(k)\top} s^{(k)} \leq 10^{-8}, \quad \text{skip the BFGS update.}$$

This simple check prevents dividing by (near) zero, preserves positive-definiteness of the Hessian estimate, and immediately stops the pathological growth in subsequent iterates—eliminating the NaNs and restoring reliable convergence.

Some of the difficulties in observing larger clusters is that the initializations start with very high energy because of there being a higher chance of points spawning close to each other, and it's possible that they would converge to energy of 0 (by pushing all the atoms extremely far apart from each other). Some other difficulties within this problem include that depending on random initializations, both algorithms may push toward a local minimum with energy 0 because points happened to be initilized along an axis. Some of the measures to reduce this variability was using different initial guesses than uniform random. For example, for most cases, we allowed the initial guess for points to be equally-spaced on the surface of a unit sphere. This produced mostly* good results for small values of $N$.

## 6 Algorithm Pseudo-code

We referenced the textbook for help with generating the pseudo algorithm (Ascher et al., 2011). We also referenced two external references for implementing Strong Wolfe Condition in Algorithm 2 (Ghosh, 2021; Nocedal & Wright, 2006).

**Algorithm 1** Gradient Descent

**Require:** initial $X^{(0)}$, parameters $\alpha_0, \beta, c$, tolerances $\epsilon_g, \epsilon_E$, max iter $K$

**for** $k = 0$ to $K - 1$ **do**
  compute $g^{(k)} = \nabla E(X^{(k)})$
  $\alpha \leftarrow \alpha_0$
  **while** $E(X^{(k)} - \alpha\, g^{(k)}) > E(X^{(k)}) - c\,\alpha\, \|g^{(k)}\|^2$ **do**
    $\alpha \leftarrow \beta\, \alpha$
  **end while**
  $X^{(k+1)} \leftarrow X^{(k)} - \alpha\, g^{(k)}$
  **if** $\Delta E \le \epsilon_E$ or $\|g^{(k)}\| \le \epsilon_g$ **then**
    break ▷ delta-energy, gradient-norm guard
  **end if**
**end for**

**Algorithm 2** BFGS

**Require:** initial $X^{(0)}$, free vector $x^{(0)}$, $H_0^{-1} = I$, parameters $\alpha_0, \beta, c$, tolerances $\epsilon_g, \epsilon_E$, max iter $K$

**for** $k = 0$ to $K - 1$ **do**
  compute full gradient $g^{(k)} = \nabla E(X^{(k)})$
  extract free gradient $g_{\text{free}}^{(k)}$
  **if** $\|g_{\text{free}}^{(k)}\| \leq \epsilon_g$ **then**
    break ▷ gradient-norm guard
  **end if**
  $s^{(k)} \leftarrow -H_k^{-1}\, g_{\text{free}}^{(k)}$
  Strong Wolfe Condition line search to find $\alpha_k$
  $x^{(k+1)} \leftarrow x^{(k)} + \alpha_k\, s^{(k)}$
  assemble $X^{(k+1)}$ from free $x^{(k+1)}$, reenable gradient tracking
  compute $g_{\text{free}}^{(k+1)}$
  $y^{(k)} \leftarrow g_{\text{free}}^{(k+1)} - g_{\text{free}}^{(k)}$
  $w^{(k)} \leftarrow \alpha_k s^{(k)}$
  **if** $(y^{(k)})^T w^{(k)} < 10^{-8}$ **then**
    skip BFGS update
  **else**
    $\rho = 1/((y^{(k)})^T w^{(k)})$
    $V = I - \rho\, w^{(k)} (y^{(k)})^T$
    $H_{k+1}^{-1} = V\, H_k^{-1}\, V^T + \rho\, w^{(k)} (w^{(k)})^T$
  **end if**
  $\Delta E = |E(X^{(k+1)}) - E(X^{(k)})|$
  **if** $\Delta E \leq \epsilon_E$ **then**
    break ▷ delta-energy guard
  **end if**
**end for**

# 7 References


1. The Cambridge Cluster Database. D. J. Wales, J. P. K. Doye, A. Dullweber, M. P. Hodges, F. Y. Naumkin F. Calvo, J. Hernández-Rojas and T. F. Middleton, URL http://www-wales.ch.cam.ac.uk/CCD.html.

2. Ghosh, I. (2021, July 29). Introduction to mathematical optimization. Chapter 4 Line Search Descent Methods. https://indrag49.github.io/Numerical-Optimization/line-search-descent-methods.html

3. Nocedal, J., & Wright, S. J. (2006). Numerical optimization. Springer Science+Business Media, LLC.: Springer e-books.

4. Ascher, U. M., & Greif, C. (2011). A first course in numerical methods. SIAM, Society for Industrial and Applied Mathematics.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import torch
import generator
import plotly.graph_objects as go
from itertools import combinations
from scipy.spatial.transform import Rotation as R
from sklearn.decomposition import PCA
```

In [2]:
```
def V(X, i, j):
    r_ij = X[i] - X[j]
    r = torch.norm(r_ij).pow(-6)
    return r.pow(2) - 2 * r

def center(X):
    Y = X - X[0]
    return Y

def system_potential(X):
    N, _ = X.shape
    energy = 0.0
    Y = center(X)
    for i in range(N - 1):
        for j in range(i + 1, N):
            energy += V(Y, i, j)
    return energy

def line_search(X, energy, g, alpha_0=1.0, factor=0.5, c=1e-4, max_iter=10, tol=1e-8):
    alpha = alpha_0
    dot = torch.sum(-g * g).item()
    for _ in range(max_iter):
        X_k = X - alpha * g
        energy_new = system_potential(X_k)

        if energy_new <= energy + c * alpha * dot:
            break
        alpha *= factor
        if alpha < tol:
            break
    return alpha

def gd_solve(natoms, lr=0.008, energy_tol=1e-8, tol=1e-8, max_iter=2001, debug=True, deb
    X = gen(natoms).clone().requires_grad_().to('cuda' if torch.cuda.is_available() else
    if save_initial:
        orig = X.clone()
    if track_energy:
        nrg_list = []

    for _iter_ in range(max_iter):
        energy = system_potential(X)
        if track_energy:
            nrg_list.append(energy.detach())
        g = torch.autograd.grad(energy, X)[0]

        g_norm = g.norm().item()
        if g_norm < tol:
            if debug:
```

```
                print(f"Converged on step {_iter_}, energy: {energy.item():.6f}, gradien
            break

        with torch.no_grad():
            X -= line_search(X, energy, g) * g
            X.requires_grad_()

        energy_new = system_potential(X)

        if abs(energy_new - energy) < energy_tol:
            if debug:
                print(f"Converged on step {_iter_}, energy: {energy_new.item():.6f}, gra
            break

        if debug and _iter_ % debug_rate == 0:
            print(f"Step {_iter_}, energy: {energy.item():.6f}, gradient norm: {g_norm:.
    if save_initial:
        return orig.detach(), center(X.detach()), energy
    if track_energy:
        return torch.tensor(nrg_list), energy
    return center(X.detach()), energy
```

In [3]:

```
def plot_3d_points(points, energy):
    points_np = center(points).numpy()
    N,_ = points_np.shape
    x = points_np[:, 0]
    y = points_np[:, 1]
    z = points_np[:, 2]
    fig = go.Figure()
    for i, j in combinations(range(len(points_np)), 2):
        edge_len = np.linalg.norm(points_np[i] - points_np[j])
        V_ij = V(points, i, j)
        fig.add_trace(go.Scatter3d(
            x=[points_np[i, 0], points_np[j, 0]],
            y=[points_np[i, 1], points_np[j, 1]],
            z=[points_np[i, 2], points_np[j, 2]],
            mode='lines',
            line=dict(color='rgb(173, 216, 230, 0.4)', width=8),
            showlegend=False,
            hovertext=f"Length: {edge_len:.2e}, Contributes {V_ij:.2e}J",
            hoverinfo="text",
        ))
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
        hovertext=[f"Point {i + 1}" for i in range(N)],
        marker=dict(size=8, color='red'),
        showlegend=False
    ))
    fig.update_layout(
        scene=dict(
            xaxis=dict(showgrid=False, zeroline=False, showticklabels=False, title=''),
            yaxis=dict(showgrid=False, zeroline=False, showticklabels=False, title=''),
            zaxis=dict(showgrid=False, zeroline=False, showticklabels=False, title=''),
        ),
        title=f"{N}-Atom Configuration: {energy:.4f}J",
        width=600,
        height=400,
        plot_bgcolor='rgba(0,0,0,0)',
```

```
        paper_bgcolor='rgba(0,0,0,0)',
    )
    fig.update_scenes(xaxis_visible=False, yaxis_visible=False,zaxis_visible=False)
    fig.show()
```

In [4]:
```
for atoms in range(2,15):
    config = gd_solve(atoms, debug=False)
    plot_3d_points(*config)
```

2-Atom Configuration: -1.0000J

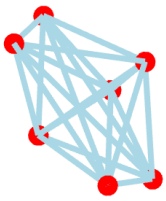
3-Atom Configuration: -3.0000J

4-Atom Configuration: -6.0000J

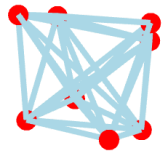
5-Atom Configuration: -9.1039J

6-Atom Configuration: -12.7121J

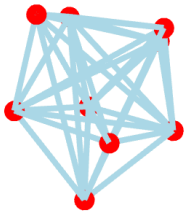
7-Atom Configuration: -16.5054J



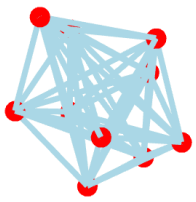
8-Atom Configuration: -19.7653J



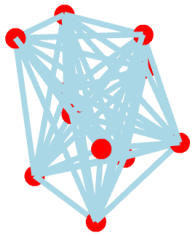
9-Atom Configuration: -23.2698J



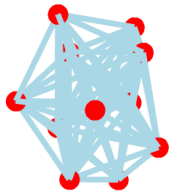
10-Atom Configuration: -26.7717J



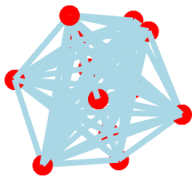
11-Atom Configuration: -32.7660J



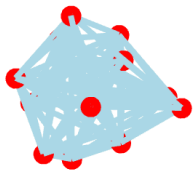
12-Atom Configuration: -34.3944J



13-Atom Configuration: -40.7585J



14-Atom Configuration: -42.5793J



In [5]:
```python
# Verify that line search works right

X = generator.uniform_sphere(3).clone().requires_grad_(True)
for i in range(20):
    E = system_potential(X)
    g = torch.autograd.grad(E, X)[0]
    p = -g.view(-1)
    a = line_search(X, E, g, alpha_0=0.1)   # try 0.1 or whatever lr
    with torch.no_grad():
        X = X + a * p.view_as(X)
        X = X - X[0]
    X.requires_grad_(True)
    print(f"Iter {i:2d}: E = {E.item():.6f}, ‖grad‖ = {g.norm().item():.2e}, a = {a:.3f}
```

```
Iter  0: E = -0.221921, ‖grad‖ = 7.62e-01, a = 0.100
Iter  1: E = -0.289849, ‖grad‖ = 1.04e+00, a = 0.100
Iter  2: E = -0.424871, ‖grad‖ = 1.61e+00, a = 0.100
Iter  3: E = -0.803233, ‖grad‖ = 3.29e+00, a = 0.100
Iter  4: E = -1.279765, ‖grad‖ = 3.61e+01, a = 0.006
Iter  5: E = -1.635842, ‖grad‖ = 6.56e+00, a = 0.025
Iter  6: E = -2.858243, ‖grad‖ = 5.87e+00, a = 0.013
Iter  7: E = -2.863041, ‖grad‖ = 9.64e+00, a = 0.006
Iter  8: E = -2.927801, ‖grad‖ = 4.59e+00, a = 0.006
Iter  9: E = -2.999994, ‖grad‖ = 4.41e-02, a = 0.006
Iter 10: E = -3.000000, ‖grad‖ = 1.52e-02, a = 0.006
Iter 11: E = -3.000000, ‖grad‖ = 5.25e-03, a = 0.006
Iter 12: E = -3.000000, ‖grad‖ = 1.80e-03, a = 0.025
Iter 13: E = -3.000000, ‖grad‖ = 7.38e-03, a = 0.006
Iter 14: E = -3.000000, ‖grad‖ = 2.58e-03, a = 0.013
Iter 15: E = -3.000000, ‖grad‖ = 4.34e-03, a = 0.006
Iter 16: E = -3.000000, ‖grad‖ = 1.52e-03, a = 0.025
Iter 17: E = -3.000000, ‖grad‖ = 6.68e-03, a = 0.006
Iter 18: E = -3.000000, ‖grad‖ = 2.33e-03, a = 0.013
Iter 19: E = -3.000000, ‖grad‖ = 3.95e-03, a = 0.006
```

In [6]:
```
def strong_wolfe_line_search(X, energy, g, p,
                             alpha0=1.0, c1=1e-4, c2=0.9,
                             max_iter=20):
    def phi(a):
        Xt = X + a * p
        return system_potential(Xt).item()
    def der_phi(a):
        Xt = X + a * p
        Xt = Xt.clone().detach().requires_grad_(True)
        Et = system_potential(Xt)
        grad_t = torch.autograd.grad(Et, Xt)[0]
        return (grad_t.view(-1) @ p.view(-1)).item()

    alpha_prev = 0.0
    phi0       = energy
    derphi0    = (g.view(-1) @ p.view(-1)).item()
    alpha = alpha0
    phi_prev = phi0

    for i in range(max_iter):
        phi_a = phi(alpha)
        if (phi_a > phi0 + c1 * alpha * derphi0) or (i > 0 and phi_a >= phi_prev):
            # need to zoom between alpha_prev and alpha
            return zoom(alpha_prev, alpha, phi, der_phi, phi0, derphi0, c1, c2)
        derphi_a = der_phi(alpha)
        if abs(derphi_a) <= -c2 * derphi0:
            return alpha
        if derphi_a >= 0:
            # curvature condition violated: derivative has changed sign
            return zoom(alpha, alpha_prev, phi, der_phi, phi0, derphi0, c1, c2)
        alpha_prev, phi_prev = alpha, phi_a
        alpha = alpha * 2.0  # increase alpha and keep searching

    return alpha  # fallback

def zoom(a_lo, a_hi, phi, der_phi, phi0, derphi0, c1, c2):
    for j in range(20):
        # interpolate midpoint
        a_j = 0.5 * (a_lo + a_hi)
        phi_aj = phi(a_j)
        if (phi_aj > phi0 + c1 * a_j * derphi0) or (phi_aj >= phi(a_lo)):
            a_hi = a_j
        else:
            derphi_aj = der_phi(a_j)
            if abs(derphi_aj) <= -c2 * derphi0:
                return a_j
            if derphi_aj * (a_hi - a_lo) >= 0:
                a_hi = a_lo
            a_lo = a_j
    return a_j
```

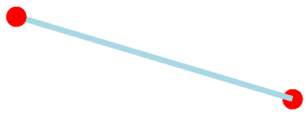
In [7]:
```
def bfgs_step(X, energy, g, H_inv, alpha=1.0, max_iter=10, tol=1e-8):
    N, _ = X.shape

    # flatten & slice out the free variables
    g_full = g.view(-1)
    g_free = g_full[3:]
    # print("||g_full|| =", g_full.norm().item())
```

```
    # print("||g_free|| =", g_free.norm().item())
    # print("H_inv diag:", torch.diag(H_inv)[:6])
    # print("H_inv.shape =", H_inv.shape)
    # print("g_free.shape =", g_free.shape)

    # descent dir in the free subspace
    s_free = - H_inv @ g_free

    p = torch.zeros_like(g_full, device=g_full.device)
    p[3:] = s_free
    p = p.view_as(X)

    # line search (still uses the full X and full g)
    alpha = strong_wolfe_line_search(X, energy, g, p)

    # take a step in the free variables
    x_full = X.clone().detach().view(-1)
    x_free = x_full[3:]

    # print("||s_free|| =", s_free.norm().item())
    # print("alpha =", alpha)
    # print("x_free (first 5) =", x_free[:5])
    x_free_new = x_free + alpha * s_free
    # print("x_free_new (first 5) =", x_free_new[:5])
    # print("s = x_free_new - x_free (first 5) =", (x_free_new - x_free)[:5])

    # reassemble X_new
    x_full[3:] = x_free_new
    X_new = x_full.view(N,3).requires_grad_(True)

    # eval new energy & gradient
    energy_new = system_potential(X_new)
    g_new_full = torch.autograd.grad(energy_new, X_new)[0]
    g_new = g_new_full.view(-1)
    g_new_free = g_new[3:]

    # BFGS inverse-Hessian update on the free subspace
    w = alpha * s_free
    y = g_new_free - g_free
    # print (f"y={y}")
    # print (f"s={s}")
    # print (f"ys={y @ s}")
    ys = (y @ w).item()
    # print(f"||s|| = {w.norm().item():.2e}, ||y|| = {y.norm().item():.2e}, y^T s = {ys:


    if ys <= 1e-24:
        H_inv_new = H_inv.clone()
        valid = True
    else:
        #print ("does this run")
        rho = 1.0 / ys
        I   = torch.eye(H_inv.shape[0], device=H_inv.device)
        V   = I - rho * torch.outer(w, y)
        with torch.no_grad():
            H_inv_new = V @ H_inv @ V.T + rho * torch.outer(w, w)
        valid = True

    return X_new, energy_new, g_new_full, H_inv_new, alpha, valid
```

```
def bfgs(natoms, lr=0.008, g_tol=1e-8, energy_tol=1e-8, max_iter=2001, debug=True, debug
    X = gen(natoms).clone().requires_grad_().to('cuda' if torch.cuda.is_available() else
    if save_initial:
        orig = X.clone()
    N, _ = X.shape
    H_inv = torch.eye(3 * (N - 1), dtype=torch.float64).to(X.device)

    X = X.clone().detach().requires_grad_(True) # make sure autograd returns valid

    if track_energy:
        nrg_list = [system_potential(X)]
    for _iter_ in range(max_iter):
        energy = system_potential(X)
        g = torch.autograd.grad(energy, X)[0]

        X_new, energy_new, g_new, H_inv, alpha, valid = bfgs_step(X, energy, g, H_inv, a

        if track_energy and valid:
            nrg_list.append(energy_new.detach())

        if abs(energy_new - energy) < energy_tol:
            if debug:
                print(f"Converged on step {_iter_}, energy: {energy_new.item():.6f}, gra
            break

        g_norm = g.norm().item()
        if g_norm < g_tol:
            if debug:
                print(f"Converged on step {_iter_}, energy: {energy.item():.6f}, gradien
            break

        if debug and _iter_ % debug_rate == 0:
            print(f"Step {_iter_}, energy: {energy_new.item():.6f}, gradient norm: {g_no

        X, energy = X_new, energy_new

    if save_initial:
        return orig.detach(), X.detach(), energy_new
    if track_energy:
        return torch.tensor(nrg_list), energy_new

    return X.detach(), energy_new
```

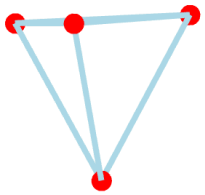
In [8]:
```
for atoms in range(2,10):
    config = bfgs(atoms, debug=False, g_tol=1e-8, energy_tol=1e-8, debug_rate=20)
    plot_3d_points(*config)
```

2-Atom Configuration: -1.0000J

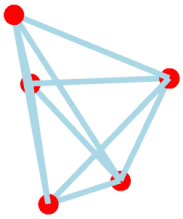


3-Atom Configuration: -3.0000J

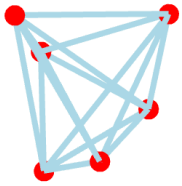
4-Atom Configuration: -6.0000J



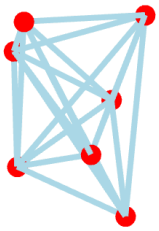
5-Atom Configuration: -9.1039J



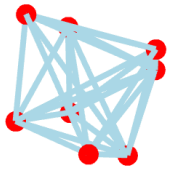
6-Atom Configuration: -12.3029J



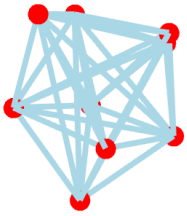
7-Atom Configuration: -15.5331J



8-Atom Configuration: -19.7653J



9-Atom Configuration: -23.2698J

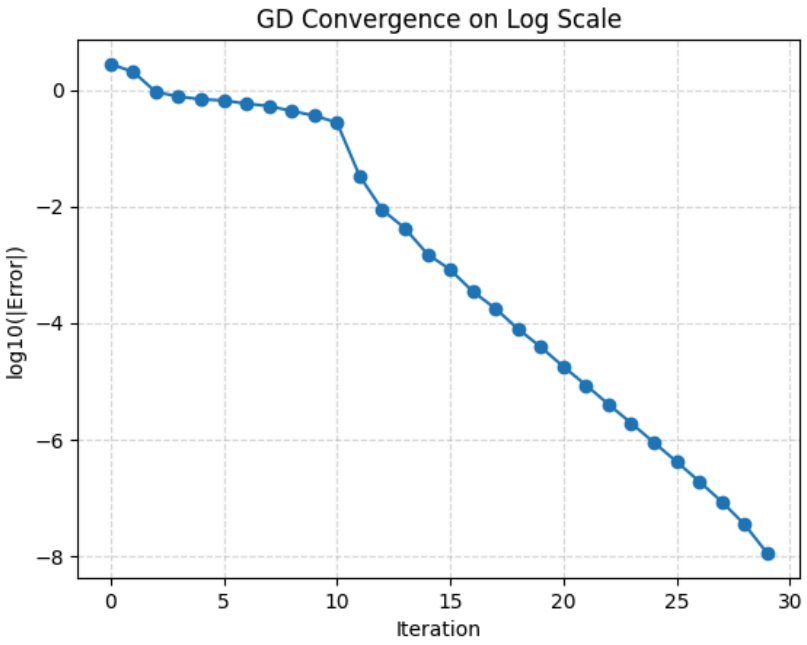


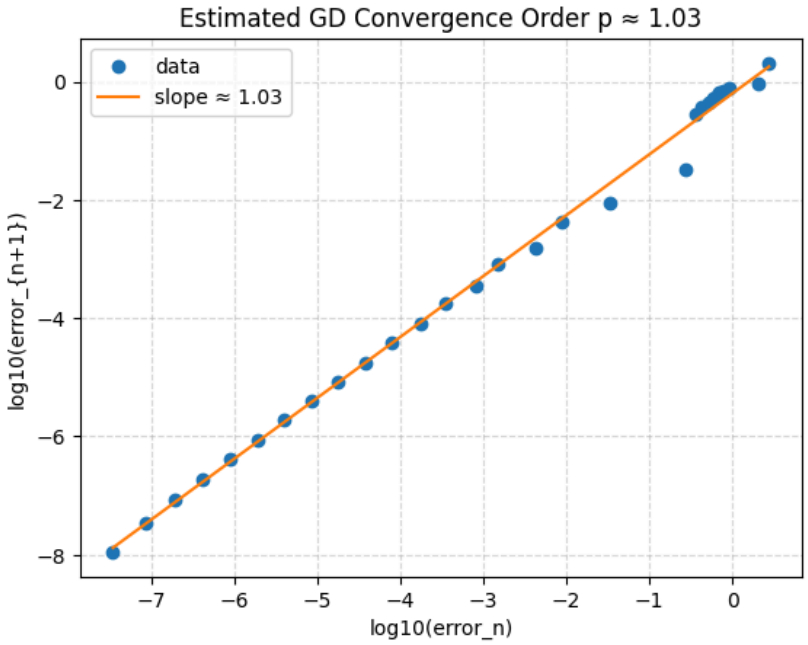
In [9]:
```python
def plot_convergence_order(energies, final_energy, label="", show=True):
    """
    energies: 1D tensor [E0, E1, ..., E_N]
    final_energy: scalar tensor E*

    Plots (log10 e_n, log10 e_{n+1}) with a linear fit whose slope ≈ order p.
    """
    # 1) drop last energy
    E = energies[:-1]
    # 2) absolute errors
```

```
    errs = (E - final_energy).abs()
    # 3) log10 errors, set zeros to -inf
    log_err = torch.where(errs > 0,
                          errs.log10(),
                          torch.full_like(errs, float("-inf")))
    # 4) keep only finite entries
    mask = torch.isfinite(log_err)
    log_err = log_err[mask]
    # 5) x = log_err[:-1], y = log_err[1:]
    x = log_err[:-1]
    y = log_err[1:]
    # 6) compute slope p and intercept a in torch
    xm = x.mean()
    ym = y.mean()
    num = ((x - xm) * (y - ym)).sum()
    den = ((x - xm)**2).sum()
    p = num / den              # slope
    a = ym - p * xm            # intercept

    # 7) convert to NumPy for plotting
    x_np = x.detach().numpy()
    y_np = y.detach().numpy()
    p_val = p.item()
    a_val = a.item()
    if show:
        # 8) plot
        plt.figure()
        plt.plot(x_np, y_np, "o", label="data")
        plt.plot(x_np, p_val * x_np + a_val, "-", label=f"slope ≈ {p_val:.2f}")
        plt.xlabel("log10(error_n)")
        plt.ylabel("log10(error_{n+1})")
        plt.title(f"Estimated {label}Convergence Order p ≈ {p_val:.2f}")
        plt.grid(True, linestyle="--", alpha=0.5)
        plt.legend()
        plt.show()
    return p_val

def plot_convergence(energies, final_energy, label=""):
    """
    Plots the convergence of absolute errors on a log10 scale,
    ignoring the last entry in the energies tensor.

    energies: 1D torch.Tensor of computed energies [E0, E1, ..., EN]
    final_energy: scalar torch.Tensor (or float castable) E*
    """
    # drop the last computed energy
    E = energies[:-1]
    errors = (E - final_energy).abs()

    # compute log10(errors), putting -inf where error is zero
    log_errors = torch.where(
        errors > 0,
        errors.log10(),
        torch.full_like(errors, float("-inf"))
    )

    # move to CPU+NumPy for plotting
    iters = torch.arange(log_errors.size(0)).cpu().numpy()
    log_e = log_errors.detach().numpy()
```

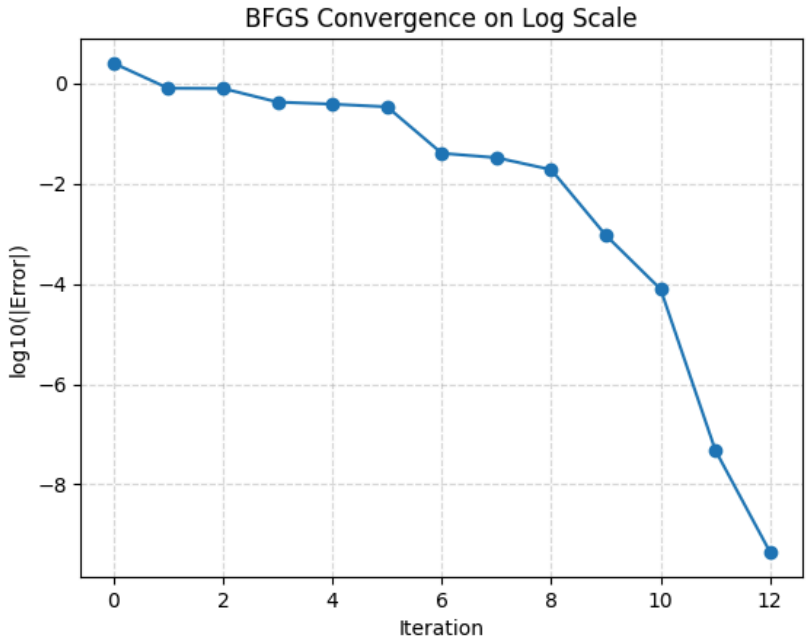
```
    plt.figure()
    plt.plot(iters, log_e, marker='o')
    plt.xlabel('Iteration')
    plt.ylabel('log10(|Error|)')
    plt.title(f'{label}Convergence on Log Scale')
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.show()

    plot_convergence_order(energies, final_energy, label)
```

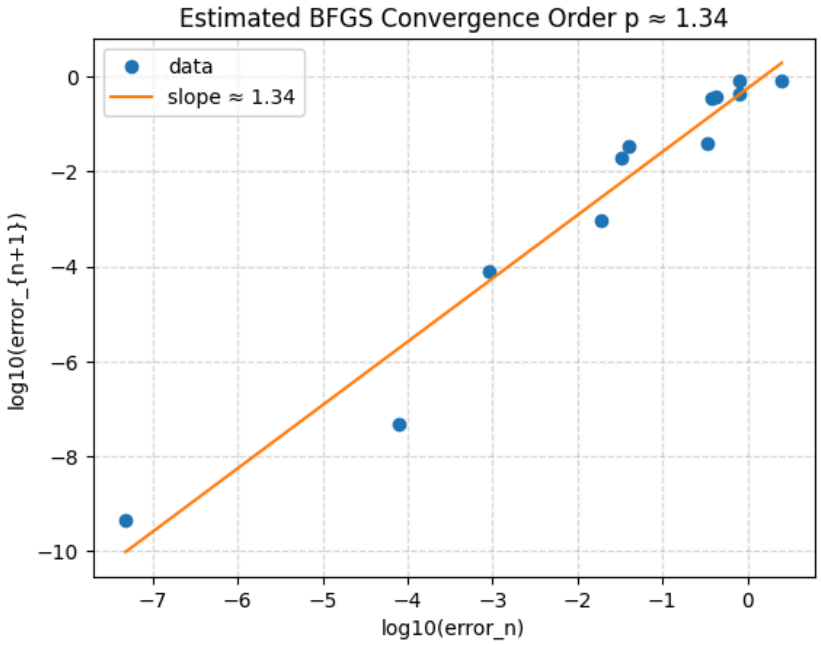
In [10]:
```
plot_convergence(*(gd_solve(3, track_energy=True, debug=False, gen=generator.uniform_sph
```






In [11]:
```
plot_convergence(*(bfgs(3, track_energy=True, debug=False, gen=generator.init_pos)), lab
```





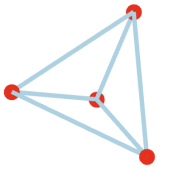

In [14]:
```
p_vals = []

for i in range(100):
    p_val = plot_convergence_order(*(bfgs(3, track_energy=True, debug=False, gen=generat
    p_vals.append(p_val)

print (np.nanmean(p_vals))
```

```
1.4892628612954626
```

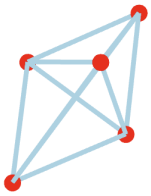
In [13]:
```
p_vals = []

for i in range(100):
    p_val = plot_convergence_order(*(gd_solve(3, track_energy=True, debug=False, gen=gen
    p_vals.append(p_val)

print (np.nanmean(p_vals))
```

```
1.0548082989579861
```

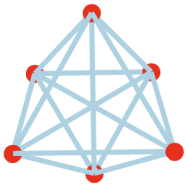
In [ ]:
```

```



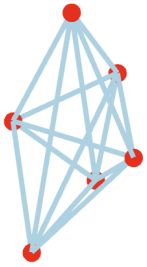
(a) N=4



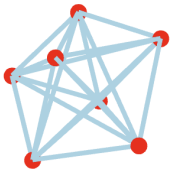
(b) N=5



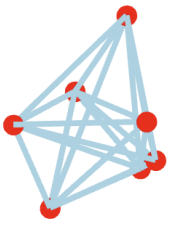
(c) N=6 (GD)



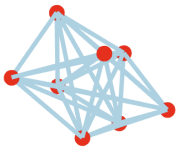
(d) N=6 (BFGS)



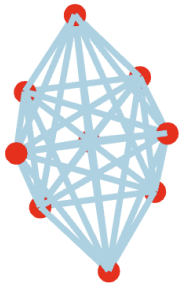
(e) N=7 (GD)



(f) N=7 (BFGS)



(g) N=8



(h) N=9

Figure 5: Geometries for $N = 4, 5, 6, 7, 8, 9$.