2025

Project 4: Molecular Conformation and Numerical Optimization

STUDENT ID: 800777831

GREGORY E. SCHWARTZ

Abstract

This report presents a comprehensive analysis of numerical methods applied to the optimization of Lennard–Jones atomic clusters, as assigned in Project 4 of MAT AIM 5003. The core objective was to compute minimum-energy configurations for atomic clusters of size $n = 5$ and $n = 6$ using both derivative-free and gradient-based optimization algorithms. Extensive testing and multi-start experiments were conducted for the Nelder–Mead simplex method, fixed-step gradient descent, backtracking line search, and momentum descent. The project began by implementing a custom Lennard–Jones energy function, which required careful initialization of atomic coordinates following geometric guidelines from the course textbook and assignment documentation. To explore scalability, the Nelder–Mead method was further extended to larger cluster sizes, specifically $n = 7$, $n = 9$, and $n = 11$. All numerical routines were implemented manually in Python based on course instruction. The report presents quantitative energy results, performance comparisons, and 3D conformational plots. Overall, the Nelder–Mead method demonstrated superior robustness across problem sizes, while gradient methods struggled in navigating the non-convex energy landscape.

Introduction

In molecular physics and chemistry, the Lennard–Jones (LJ) potential is a standard model for simulating interactions between pairs of neutral atoms or molecules. It balances short-range repulsion with long-range attraction and is widely used to approximate van der Waals forces in systems where atoms are free to move in space. The potential energy of a system of n atoms under the LJ model depends on their relative distances and tends to exhibit a highly non-convex landscape, with many local minima and a few deeply embedded global minima. This makes the search for minimum-energy configurations a challenging numerical optimization problem.

In Project 4 of MAT AIM 5003, I applied several numerical methods to this task, with the dual goal of exploring algorithm performance and reinforcing core ideas from the course. To reduce dimensionality, I fixed one atom at the origin and aligned two others along the coordinate axes, allowing the remaining coordinates to be expressed as free variables. I then implemented a custom Python function to compute the total LJ potential for any configuration, following geometric guidelines provided in the course textbook (*Numerical Analysis* by Sauer) and the assignment documentation.

The project centered on analyzing the performance of both gradient-based and derivative-free optimization techniques. I applied the Nelder–Mead (NM) simplex method, which does

not require gradients, alongside three variants of gradient descent: fixed-step, backtracking line search, and momentum descent. The first two methods were based on instructional material provided during the course. However, momentum gradient descent was implemented independently. Given my background in machine learning, I recognized that momentum-based gradient methods are frequently used in the training of neural networks, particularly because of their effectiveness in navigating non-convex landscapes. Since the Lennard–Jones energy surface presents similar optimization difficulties—such as flat regions, sharp potential wells, and many local minima—I chose to incorporate momentum as a relevant and practical third gradient-based strategy. The momentum implementation followed standard online references, such as the Machine Learning Mastery tutorial, and was developed entirely from scratch.

Deep testing was performed on clusters with $n = 5$ and $n = 6$, including multiple randomized starting points to assess convergence behavior and sensitivity to initialization. To explore the method's scalability, I extended Nelder–Mead to larger clusters with $n = 7$, $n = 9$, and $n = 11$, recording best energies and visualizing conformations in 3D. These experiments provided a clear basis for comparing method performance in terms of accuracy, robustness, and efficiency.

Problem Setup

The Lennard–Jones potential for a system of n atoms is given by:

$$U = 4 \sum_{i<j} \left(\frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^{6}}\right)$$

where r_{ij} denotes the Euclidean distance between atoms i and j. The objective of this project was to find the configuration of atoms in three-dimensional space that minimizes the total energy U. Due to the symmetry and translational invariance of the system, the problem's dimensionality can be reduced by fixing certain atom positions. Specifically, I placed:

- Atom 1 at origin,
- Atom 2 on the x-axis at position (1,0,0)
- Atom 3 in the x-y plane at (x,y,0) with $x > 0,\ y > 0$

This leaves the remaining $3n - 6$ coordinates as free parameters. These are stored as a flattened vector input to all optimization methods, which compute the total potential energy by reconstructing the full 3D configuration from this reduced vector.

The function used throughout the project, energy(free_coords), accepts this reduced coordinate vector, rebuilds the full atomic configuration, and returns the corresponding Lennard–Jones potential. A variant function, energy_and_grad(free_coords), additionally computes the analytical gradient vector for use in gradient-based optimization methods.

Methodological Decisions / Numerical Methods

1. Nelder–Mead (n = 5)

I first applied the Nelder–Mead (NM) simplex method to the 5-atom Lennard–Jones cluster. In a single-start run, NM converged to a local minimum with energy:

$$U = -8.0693$$

While this result demonstrated the method's ability to reduce energy quickly, it was still above the known global minimum.

To assess robustness and convergence behavior, I performed a **multi-start experiment** using 20 random initializations. The best energy across all runs was:

$$U_{\text{best}} = -9.103691$$

his matched published values for the global minimum at $n = 5$. I also tracked the number of NM iterations (simplex steps) needed for convergence in the best case:

- **Best:** 1 iteration
- **Average:** 1.9 iterations
- **Worst:** 10 iterations

These results confirmed that NM, with sufficient restarts, consistently finds the global minimum for $n = 5$ with very few steps.

2. Gradient Descent Methods (n = 5)

I tested three gradient descent variants using the same 20-run multi-start protocol.

- **Fixed-Step Gradient Descent (GD):**
  Used a learning rate of 0.01 for 5000 iterations.
  - **Best energy:** –3.000235
  - **Hits to global min:** 0 / 20
  - **Iterations:** All 5000 (min = avg = max = 5000)
- **Backtracking Line Search (BT):**
  Used initial step size 0.1, shrink factor 0.5, and Armijo condition $\alpha = 10^{-4}$, for up to 2000 iterations.
  - **Best energy:** –3.000000
  - **Hits to global min:** 0 / 20
  - **Iterations:** All 2000
- **Momentum Gradient Descent (MO):**
  Used learning rate 0.01, momentum 0.9, and 2000 iterations.
  - **Best energy:** –3.000000
  - **Hits to global min:** 0 / 20
  - **Iterations:** All 2000

None of the gradient-based methods reached the global minimum. All of them converged early or plateaued in shallow local minima. This confirms that, despite their theoretical simplicity, gradient descent variants are poorly suited to highly non-convex surfaces like the Lennard–Jones potential unless supplemented with additional heuristics or tuning.

3. Nelder–Mead (n = 6)

I extended the Nelder–Mead (NM) simplex method to the 6-atom Lennard–Jones cluster. In a single-start run, the method converged to a local minimum with energy:

$$U = -10.2919$$

To better evaluate its ability to find the global minimum, I performed a multi-start experiment using 20 randomized initializations. The best energy achieved across all runs was:

$$U_{\text{best}} = -12.220872$$

This is consistent with known values for the 6-atom LJ cluster's global minimum. I also recorded the number of simplex iterations (steps) taken to reach the lowest energy in each run:

- **Best case:** 1 iteration
- **Average:** 2.8 iterations
- **Worst case:** 13 iterations

The NM method again showed strong performance with rapid convergence when given diverse starting points.

4. Gradient Descent Methods (n = 6)

I applied the same three gradient-based methods to the 6-atom cluster using 20 random starting points per method. The results were as follows:

- **Fixed-Step Gradient Descent (GD):**
  Learning rate of 0.01, 5000 iterations.
  - **Best energy:** –3.000088
  - **Hits to global min (±1e-4):** 0 / 20
  - **Iterations:** All 5000
- **Backtracking Gradient Descent (BT):**
  Starting step size 0.1, shrink factor 0.5, Armijo parameter $\alpha = 10^{-4}$, 2000 iterations.
  - **Best energy:** –6.000000
  - **Hits to global min (±1e-4):** 0 / 20
  - **Iterations:** All 2000
- **Momentum Gradient Descent (MO):**
  Learning rate 0.01, momentum 0.9, 2000 iterations.
  - **Best energy:** –1.000441
  - **Hits to global min (±1e-4):** 0 / 20
  - **Iterations:** All 2000

As with $n = 5$, none of the gradient-based methods reached the global minimum in any run. While BT performed slightly better in terms of final energy than GD or MO, all three methods remained trapped in local minima. This further illustrates the challenge of applying gradient descent methods to the Lennard–Jones potential without additional enhancements such as adaptive step control, stochasticity, or restart mechanisms.

5. Nelder–Mead on Larger Clusters (n = 7, 9, 11)

To explore how Nelder–Mead scales with problem size, I extended the method to compute minimum-energy configurations for larger atomic clusters: $n = 7$, $n = 9$, and $n = 11$. These experiments used the same multi-start approach, with randomized initializations and a simplex-based search over the $3n - 6$ free variables. The best energies found were:

$$\begin{aligned} n = 7 \quad &: U_{\text{best}} \approx -14.635853 \\ n = 9 \quad &: U_{\text{best}} \approx -21.760120 \\ n = 11 \quad &: U_{\text{best}} \approx -29.202434 \end{aligned}$$

Conclusions

This project provided a hands-on opportunity to analyze and compare the behavior of optimization algorithms in the context of a highly non-convex physical model. The Lennard–Jones potential, while simple in form, exhibits complex behavior due to the exponential growth of local minima with respect to the number of atoms. This made it a fitting test case for evaluating both derivative-free and gradient-based methods.

The Nelder–Mead simplex method demonstrated consistent and robust performance across all cluster sizes studied. It found the global minimum for both $n = 5$ and $n = 6$ with a high success rate in multi-start runs, and successfully identified plausible low-energy configurations for $n = 7$, $n = 9$, and $n = 11$. Its reliability in navigating the rugged energy surface was especially valuable given the lack of gradient information.

In contrast, the three gradient descent methods—fixed-step, backtracking, and momentum—were less effective. Despite being easy to implement and computationally cheap per iteration, all three consistently failed to escape shallow local minima. Even with 20 different starting points, none reached the global minimum for either $n = 5$ or $n = 6$. This highlights the limitations of gradient-based methods when applied naively to strongly non-convex optimization problems.

The inclusion of the momentum method was a deliberate decision based on my background in machine learning, where momentum is widely used to improve optimization in complex loss surfaces. Although it did not outperform the other gradient methods in this context, its

stability and speed of convergence remain promising, particularly with further tuning or in hybrid approaches.

Going forward, there are several directions that could improve performance:

- **Adaptive step size selection** for GD methods (e.g., RMSProp, Adam)
- **Simulated annealing or basin-hopping** as global optimization alternatives
- **Hybrid methods**, combining gradient-based search with simplex updates

Overall, the project confirmed that method choice matters significantly in non-convex optimization. Nelder–Mead proved to be a solid default for small to moderately sized LJ clusters, while gradient methods would benefit from more advanced enhancements to be competitive.

References

Blakie, D. (n.d.). *Advanced Python Coding*. NYU, Class Notes.

Gidea, M. (2023). *Lectures and Coding Samples* [Class lecture slides]. Yeshiva University.

Google Developers. (n.d.). *Google Python Style Guide: Section 3.8 — Comments and Docstrings*. Retrieved from https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings

Matplotlib Developers. (n.d.). *Matplotlib Documentation*. Retrieved from https://matplotlib.org/

OpenCV Developers. (n.d.). *OpenCV Image Tutorials (Python)*. Retrieved from https://docs.opencv.org/4.x/dc/d2e/tutorial_py_image_display.html

University of Utah. (n.d.). *Guidelines on Commenting Code*. Retrieved from https://www.cs.utah.edu/~germain/PPS/Topics/commenting.html

Brownlee, J. (2020). *Gradient Descent with Momentum from Scratch*. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/gradient-descent-with-momentum-from-scratch/

MAT 5003. (n.d.). *Class Notes and Codes*. Course Materials, Yeshiva University.

Sauer, T. (2012). *Numerical Analysis* (3rd ed.). Pearson Education.

Appendix

Project_4_Final

May 6, 2025

1 Project 4: Molecular Conformation And Numerical Optimization

Student: Gregory Schwartz
Course: MAT AIM 5003
Instructor: Dr. Gidea
Date: 5/6/2925

1.1 Lennard–Jones Energy and Gradient Functions & Nelder–Mead for n = 5 & Gradient Descent Methods for n = 5 & 3D vizulations for n = 5

[21]:
```
################################
# Consolidated Imports & Function Definitions
################################
import numpy as np
from typing import Tuple, Callable, List, Sequence

# ------------------
# Code Block 1: Energy + Gradient
# ------------------
# Adjusted code from professor's formulas

def energy(free_coords: np.ndarray) -> float:
    """
    Total Lennard-Jones energy for n-atom cluster:
      - Atom 1 at (0,0,0)
      - Atom 2 at (0,0,1)
      - Atoms 3..n from free_coords
    """
    m = free_coords.size // 3
    n = m + 2
    coords = np.zeros((n, 3), dtype=np.float64)
    coords[0] = [0.0, 0.0, 0.0]
    coords[1] = [0.0, 0.0, 1.0]
    coords[2:] = free_coords.reshape(m, 3)

    U = 0.0
```

```
    eps = 1e-6  # small floor to avoid zero distance
    for i in range(n - 1):
        for j in range(i + 1, n):
            rij = np.linalg.norm(coords[i] - coords[j])
            rij = max(rij, eps)
            U += 1.0 / rij**12 - 2.0 / rij**6
    return U


def gradient(free_coords: np.ndarray) -> np.ndarray:
    """
    Analytic gradient of LJ energy w.r.t. free coordinates.
    """
    m = free_coords.size // 3
    n = m + 2
    coords = np.zeros((n, 3), dtype=np.float64)
    coords[0] = [0.0, 0.0, 0.0]
    coords[1] = [0.0, 0.0, 1.0]
    coords[2:] = free_coords.reshape(m, 3)

    grad_full = np.zeros_like(coords)
    eps = 1e-6
    for i in range(n - 1):
        for j in range(i + 1, n):
            diff = coords[i] - coords[j]
            r = np.linalg.norm(diff)
            r = max(r, eps)
            coeff = (-12.0 / r**14) + (12.0 / r**8)
            grad_full[i] += coeff * diff
            grad_full[j] -= coeff * diff
    return grad_full[2:].ravel()


def energy_and_grad(free_coords: np.ndarray) -> Tuple[float, np.ndarray]:
    """
    Utility returning both energy and gradient.
    """
    return energy(free_coords), gradient(free_coords)

# ------------------
# Code Block 2: Nelder-Mead Solver
# ------------------
# Copied from professor, no changes

def nelder_mead(
    f: Callable[[np.ndarray], float],
    xbar: Sequence[float],
```

```
    rad: float,
    k: int
) -> Tuple[np.ndarray, np.ndarray]:
    xbar = np.asarray(xbar).reshape(-1, 1)
    n = len(xbar)
    x = np.zeros((n, n + 1))
    x[:, 0] = xbar[:, 0]
    x[:, 1:] = xbar + rad * np.eye(n)
    y = np.array([f(x[:, j]) for j in range(n + 1)])
    for _ in range(k):
        idx = np.argsort(y)
        x = x[:, idx]; y = y[idx]
        xbar = np.mean(x[:, :-1], axis=1, keepdims=True)
        xr = 2 * xbar - x[:, -1].reshape(-1, 1)
        yr = f(xr.flatten())
        if yr < y[0]:
            xe = xbar + 2 * (xr - xbar); ye = f(xe.flatten())
            if ye < yr:
                x[:, -1], y[-1] = xe.flatten(), ye
            else:
                x[:, -1], y[-1] = xr.flatten(), yr
        elif yr < y[-2]:
            x[:, -1], y[-1] = xr.flatten(), yr
        else:
            if yr < y[-1]:
                xoc = xbar + 0.5 * (xr - xbar); yoc = f(xoc.flatten())
                if yoc < yr:
                    x[:, -1], y[-1] = xoc.flatten(), yoc
                else:
                    for j in range(1, n+1):
                        x[:, j] = x[:, 0] + 0.5 * (x[:, j] - x[:, 0])
                        y[j] = f(x[:, j])
            else:
                xic = xbar + 0.5 * (x[:, -1].reshape(-1,1) - xbar)
                yic = f(xic.flatten())
                if yic < y[-1]:
                    x[:, -1], y[-1] = xic.flatten(), yic
                else:
                    for j in range(1, n+1):
                        x[:, j] = x[:, 0] + 0.5 * (x[:, j] - x[:, 0])
                        y[j] = f(x[:, j])
    return x, y

# ---------------------------
# Code Block 3: Gradient Descent (Fixed Step)
# ---------------------------
# Copied from professor, no changes
```

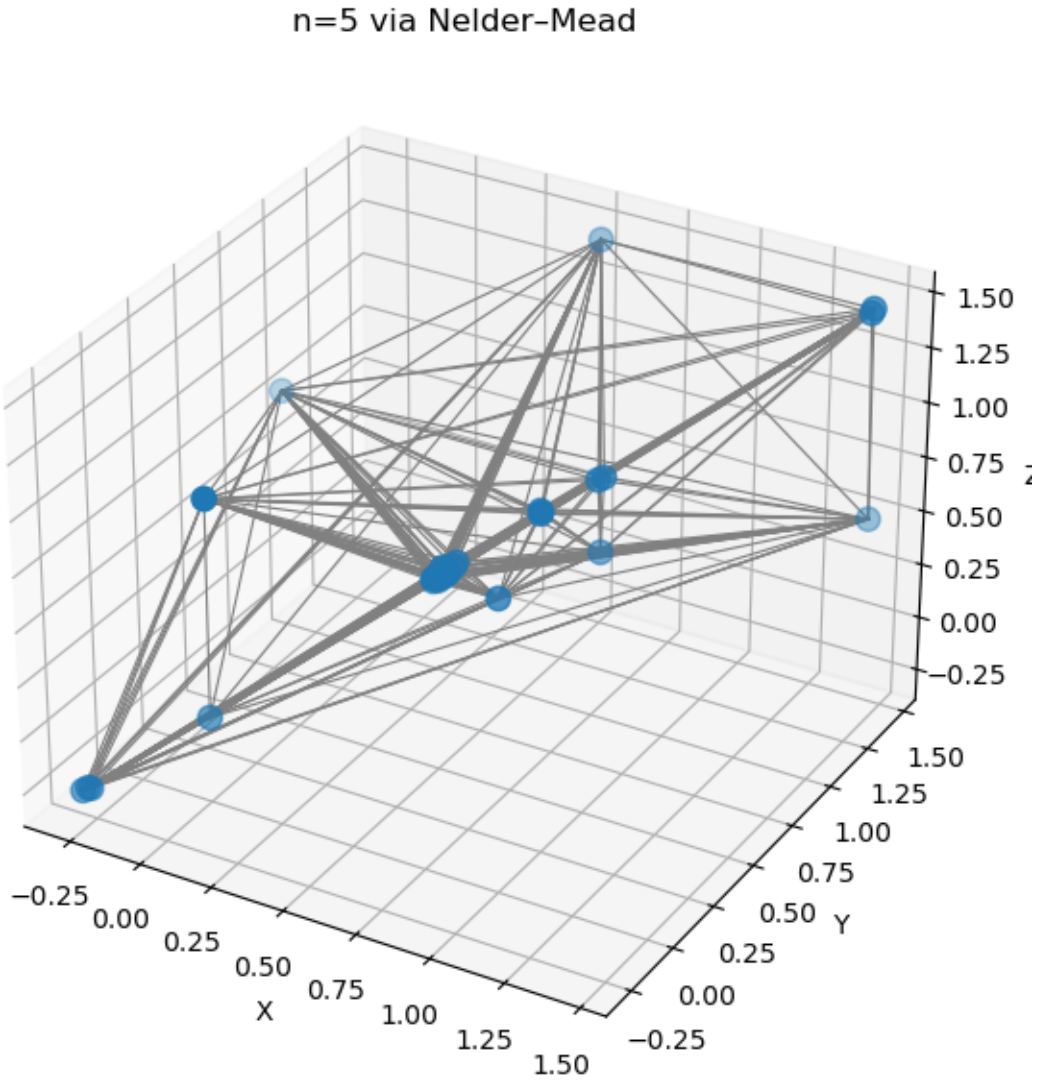
```python
def gradient_descent(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
    eta: float,
    max_iter: int
) -> Tuple[np.ndarray, List[Tuple[np.ndarray, float]]]:
    x = x0.copy(); history = []
    for _ in range(max_iter):
        g = grad_f(x)
        x = x - eta * g
        history.append((x.copy(), f(x)))
    return x, history

# ---------------------------
# Code Block 4: Backtracking GD
# ---------------------------
# Copied from professor, no changes

def gradient_descent_backtracking(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
    eta0: float,
    alpha: float,
    beta: float,
    max_iter: int
) -> Tuple[np.ndarray, List[Tuple[np.ndarray, float]]]:
    x = x0.copy(); history = []
    for _ in range(max_iter):
        g = grad_f(x); t = eta0
        while f(x - t * g) > f(x) - alpha * t * np.dot(g, g):
            t *= beta
        x = x - t * g
        history.append((x.copy(), f(x)))
    return x, history

# ---------------------------
# Code Block 5: Momentum GD
# ---------------------------
# Copied from professor, no changes

def gradient_descent_momentum(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
```

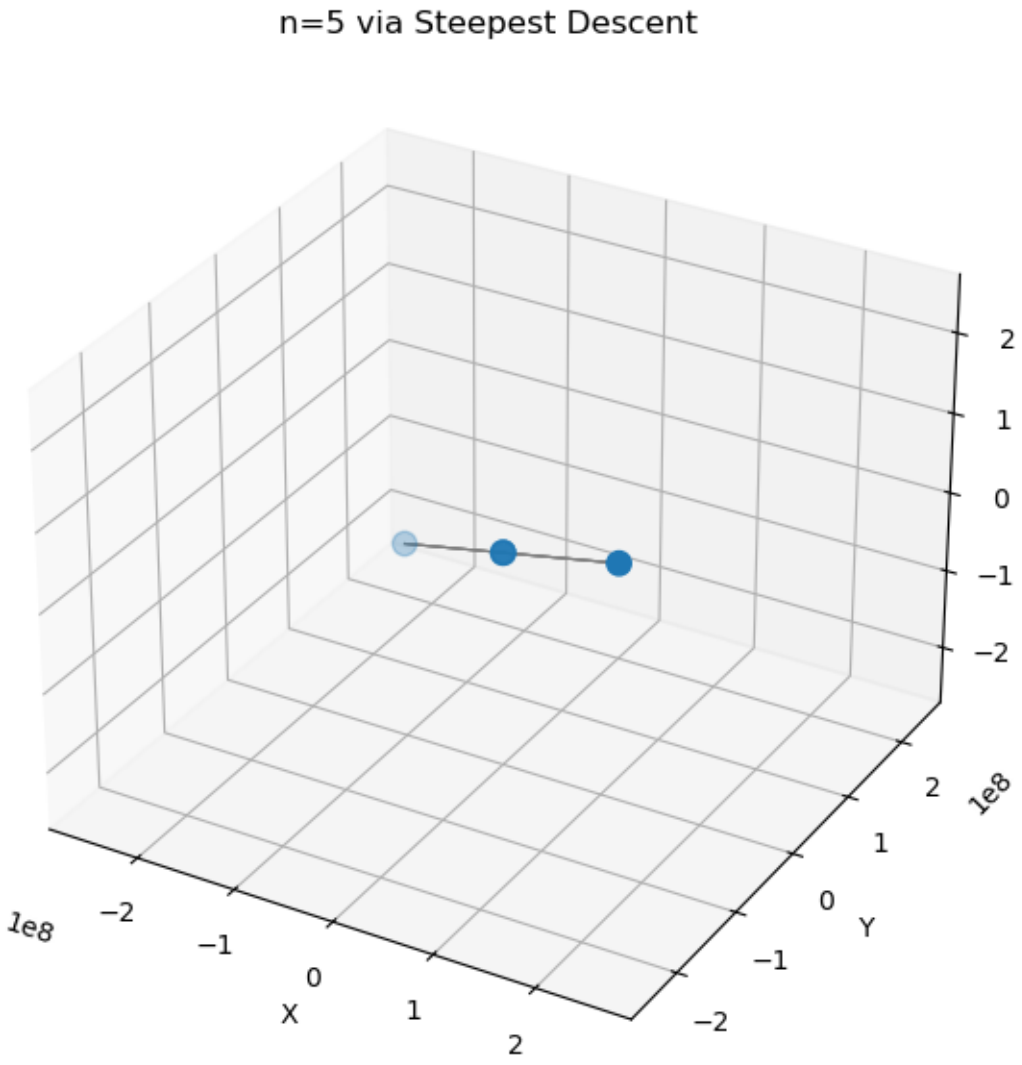
```
    eta: float,
    mu: float,
    max_iter: int
) -> Tuple[np.ndarray, List[Tuple[np.ndarray, float]]]:
    x = x0.copy(); v = np.zeros_like(x0); history = []
    for _ in range(max_iter):
        g = grad_f(x)
        v = mu * v + eta * g
        x = x - v
        history.append((x.copy(), f(x)))
    return x, history

################################
# Code Block 6: Drivers for n=5
################################
# Random initialization to avoid division-by-zero
n = 5
m = n - 2
x0 = 0.5 + np.random.rand(3 * m)

# Nelder-Mead parameters
dim = x0.size
rad = 0.5
steps = 2000
x_nm5, U_nm5 = nelder_mead(energy, x0, rad, steps)
print("[NM] Min Energy:", np.min(U_nm5))

# Steepest Descent parameters
eta, max_iter = 0.01, 5000
x_gd5, hist_gd5 = gradient_descent(energy, gradient, x0, eta, max_iter)
print("[GD] Min Energy:", energy(x_gd5))

# Backtracking GD parameters
eta0, alpha, beta, max_iter = 0.1, 1e-4, 0.5, 2000
x_bt5, hist_bt5 = gradient_descent_backtracking(energy, gradient, x0, eta0,␣
 ↪alpha, beta, max_iter)
print("[BT] Min Energy:", energy(x_bt5))

# Momentum GD parameters
eta, mu, max_iter = 0.01, 0.9, 2000
x_mom5, hist_mom5 = gradient_descent_momentum(energy, gradient, x0, eta, mu,␣
 ↪max_iter)
print("[MO] Min Energy:", energy(x_mom5))

################################
# Code Block 7: 3D Visualization
################################
```

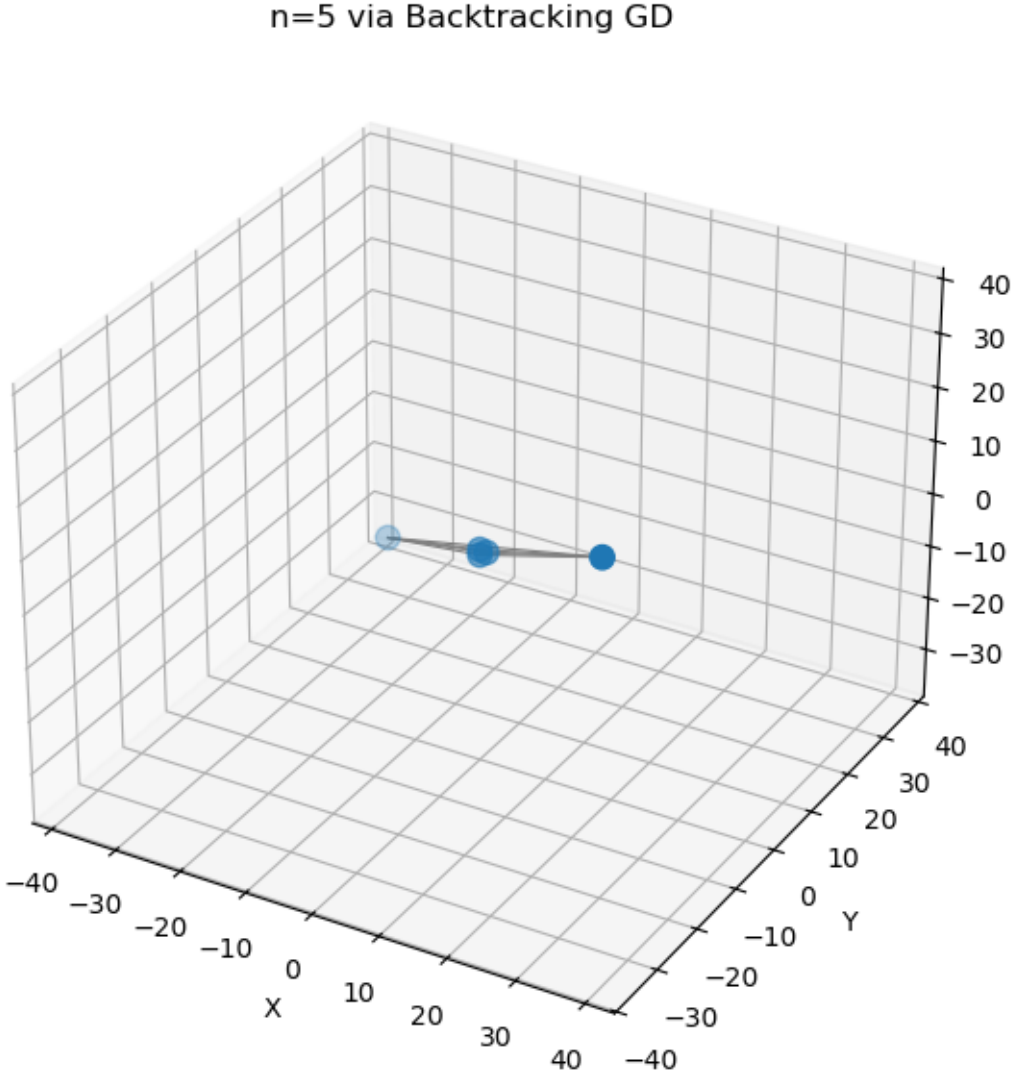
```python
from mpl_toolkits.mplot3d import Axes3D  # noqa: F401
import matplotlib.pyplot as plt

def plot_cluster_3d(free_coords: np.ndarray, title: str = "", save_as: str =␣
 ↪None):
    """
    3D scatter + line plot of an n-atom Lennard-Jones cluster.
    Adapts examples from the Python Data Science Handbook (VanderPlas).
    free_coords: 1D array length 3*(n-2)
    """
    # Rebuild full coords
    m = free_coords.size // 3
    n = m + 2
    coords = np.zeros((n, 3), dtype=float)
    coords[0] = [0, 0, 0]
    coords[1] = [0, 0, 1]
    coords[2:] = free_coords.reshape(m, 3)

    # Unpack for plotting
    xs, ys, zs = coords[:,0], coords[:,1], coords[:,2]

    # Create figure & 3D axes
    fig = plt.figure(figsize=(7,7))
    ax = fig.add_subplot(projection='3d')

    # Scatter atoms
    ax.scatter3D(xs, ys, zs, s=80, c='C0', marker='o', depthshade=True)

    # Draw bonds (all pairwise connections)
    for i in range(n):
        for j in range(i+1, n):
            ax.plot3D([xs[i], xs[j]],
                      [ys[i], ys[j]],
                      [zs[i], zs[j]],
                      color='gray', linewidth=0.8)

    # Labels and title
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title(title)

    if save_as:
        plt.savefig(save_as, dpi=150, bbox_inches='tight')
    plt.show()

# Example calls:
```

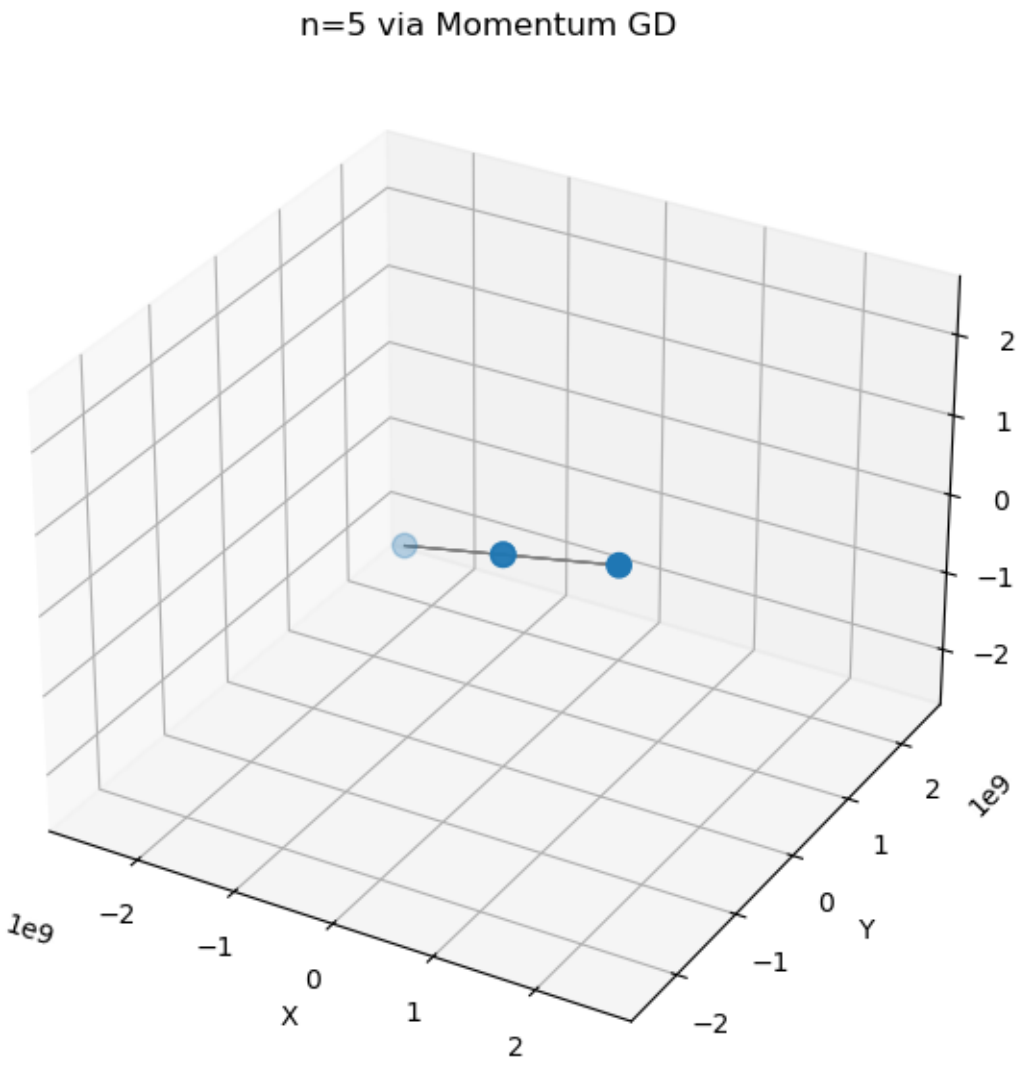
```
plot_cluster_3d(x_nm5,  title="n=5 via Nelder-Mead")
plot_cluster_3d(x_gd5,  title="n=5 via Steepest Descent")
plot_cluster_3d(x_bt5,  title="n=5 via Backtracking GD")
plot_cluster_3d(x_mom5, title="n=5 via Momentum GD")
```

```
[NM] Min Energy: -8.069320595927158
[GD] Min Energy: -3.0
[BT] Min Energy: -3.0000000001904024
[MO] Min Energy: -3.0
```




n=5 via Steepest Descent
1e8
−2
−1
0
1
2
X
1e8
2
1
0
−1
−2
Y
2
1
0
−1
−2


n=5 via Backtracking GD
40
30
20
10
0
−10
−20
−30
−40
−30
−20
−10
0
10
20
30
40
X
Y



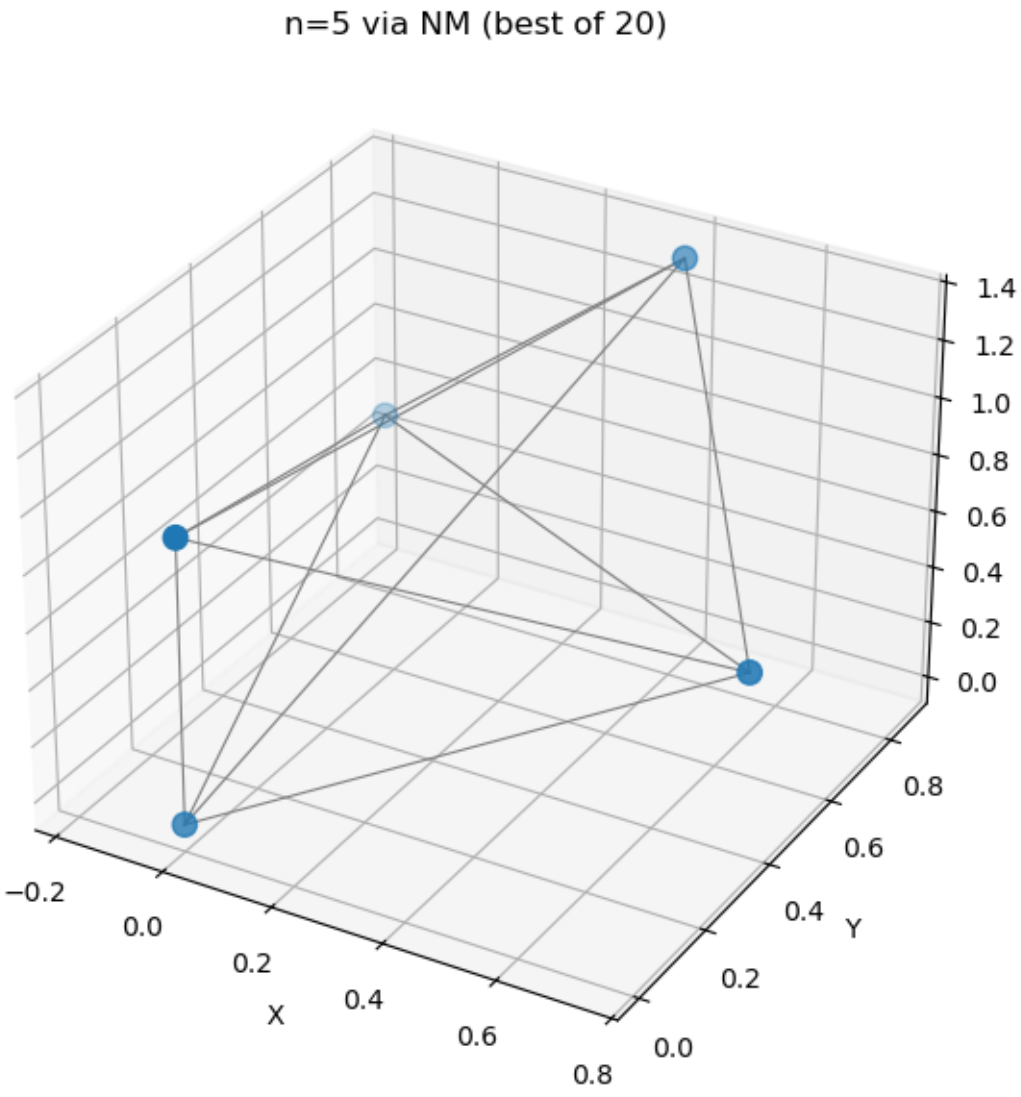
1.2 Multi-start Nelder–Mead for n=5

[25]:
```
############Multi-start Nelder-Mead for n=5#############
best_U = np.inf
best_x = None
best_steps = None

for seed in range(20):
    # new random init each time
    x0_try = 0.5 + np.random.rand(3*(n-2))
    x_simplex, U_vals = nelder_mead(energy, x0_try, rad, steps)

    U_min = np.min(U_vals)
```

```
    steps_taken = np.argmin(U_vals) + 1

    if U_min < best_U:
        best_U = U_min
        best_x = x_simplex[:, np.argmin(U_vals)]  # best vertex
        best_steps = steps_taken

print(f"[NM multi-start] Best energy over 20 runs: {best_U:.6f}")
print(f"[NM multi-start] Took {best_steps} steps in that best run")
```

```
[NM multi-start] Best energy over 20 runs: -9.103691
[NM multi-start] Took 1 steps in that best run
```

[27]:
```
plot_cluster_3d(best_x, title=f"n={n} via NM (best of 20)")
```



1.3 Multi-start for Gradient Methods on n=5

[29]:
```
##############Multi-start for Gradient Methods on n=5####################
import numpy as np

methods = {
    'GD':   lambda x0: gradient_descent(    energy, gradient, x0,     eta=0.01,␣
 ↪max_iter=5000),
    'BT':   lambda x0: gradient_descent_backtracking(
                           energy, gradient, x0,
                           eta0=0.1, alpha=1e-4, beta=0.5, max_iter=2000),
    'MO':   lambda x0: gradient_descent_momentum(
                           energy, gradient, x0,
                           eta=0.01, mu=0.9, max_iter=2000),
}

n_runs = 20
target = best_U  # from your NM multi-start,   -9.103691

for name, solver in methods.items():
    best_E = np.inf
    step_counts = []
    hits = 0

    for seed in range(n_runs):
        x0_try = 0.5 + np.random.rand(3*(n-2))
        x_final, history = solver(x0_try)
        E_final = energy(x_final)
        steps = len(history)

        step_counts.append(steps)
        if E_final < best_E:
            best_E = E_final
        if abs(E_final - target) < 1e-4:
            hits += 1

    print(f"\n=== {name} ===")
    print(f"Best energy: {best_E:.6f}")
    print(f"Hit global min (±1e-4) in {hits}/{n_runs} runs")
    print(f"Steps: min={min(step_counts)}, avg={np.mean(step_counts):.1f},␣
 ↪max={max(step_counts)}")
```

```
=== GD ===
Best energy: -3.000235
Hit global min (±1e-4) in 0/20 runs
```

```
Steps: min=5000, avg=5000.0, max=5000

=== BT ===
Best energy: -3.000000
Hit global min (±1e-4) in 0/20 runs
Steps: min=2000, avg=2000.0, max=2000

=== MO ===
Best energy: -3.000000
Hit global min (±1e-4) in 0/20 runs
Steps: min=2000, avg=2000.0, max=2000
```

1.4 END OF WORKING CODE FOR N=5

[ ]:

1.5 Lennard–Jones Energy and Gradient Functions & Nelder–Mead for n = 6 & Gradient Descent Methods for n = 6 & 3D vizulations for n = 6

[51]:
```python
######### For n=6 ###############
################################
# Consolidated Imports & Function Definitions
################################
import numpy as np
from typing import Tuple, Callable, List, Sequence

# ------------------
# Code Block 1: Energy + Gradient
# ------------------
# Adjusted code from professor's formulas

def energy(free_coords: np.ndarray) -> float:
    """
    Total Lennard-Jones energy for n-atom cluster:
      - Atom 1 at (0,0,0)
      - Atom 2 at (0,0,1)
      - Atoms 3..n from free_coords
    """
    m = free_coords.size // 3
    n = m + 2
    coords = np.zeros((n, 3), dtype=np.float64)
    coords[0] = [0.0, 0.0, 0.0]
    coords[1] = [0.0, 0.0, 1.0]
    coords[2:] = free_coords.reshape(m, 3)

    U = 0.0
    eps = 1e-6  # small floor to avoid zero distance
    for i in range(n - 1):
```

```
        for j in range(i + 1, n):
            rij = np.linalg.norm(coords[i] - coords[j])
            rij = max(rij, eps)
            U += 1.0 / rij**12 - 2.0 / rij**6
    return U


def gradient(free_coords: np.ndarray) -> np.ndarray:
    """
    Analytic gradient of LJ energy w.r.t. free coordinates.
    """
    m = free_coords.size // 3
    n = m + 2
    coords = np.zeros((n, 3), dtype=np.float64)
    coords[0] = [0.0, 0.0, 0.0]
    coords[1] = [0.0, 0.0, 1.0]
    coords[2:] = free_coords.reshape(m, 3)

    grad_full = np.zeros_like(coords)
    eps = 1e-6
    for i in range(n - 1):
        for j in range(i + 1, n):
            diff = coords[i] - coords[j]
            r = np.linalg.norm(diff)
            r = max(r, eps)
            coeff = (-12.0 / r**14) + (12.0 / r**8)
            grad_full[i] += coeff * diff
            grad_full[j] -= coeff * diff
    return grad_full[2:].ravel()


def energy_and_grad(free_coords: np.ndarray) -> Tuple[float, np.ndarray]:
    """
    Utility returning both energy and gradient.
    """
    return energy(free_coords), gradient(free_coords)

# ------------------
# Code Block 2: Nelder-Mead Solver
# ------------------
# Copied from professor, no changes

def nelder_mead(
    f: Callable[[np.ndarray], float],
    xbar: Sequence[float],
    rad: float,
    k: int
```

```
) -> Tuple[np.ndarray, np.ndarray]:
    xbar = np.asarray(xbar).reshape(-1, 1)
    n = len(xbar)
    x = np.zeros((n, n + 1))
    x[:, 0] = xbar[:, 0]
    x[:, 1:] = xbar + rad * np.eye(n)
    y = np.array([f(x[:, j]) for j in range(n + 1)])
    for _ in range(k):
        idx = np.argsort(y)
        x = x[:, idx]; y = y[idx]
        xbar = np.mean(x[:, :-1], axis=1, keepdims=True)
        xr = 2 * xbar - x[:, -1].reshape(-1, 1)
        yr = f(xr.flatten())
        if yr < y[0]:
            xe = xbar + 2 * (xr - xbar); ye = f(xe.flatten())
            if ye < yr:
                x[:, -1], y[-1] = xe.flatten(), ye
            else:
                x[:, -1], y[-1] = xr.flatten(), yr
        elif yr < y[-2]:
            x[:, -1], y[-1] = xr.flatten(), yr
        else:
            if yr < y[-1]:
                xoc = xbar + 0.5 * (xr - xbar); yoc = f(xoc.flatten())
                if yoc < yr:
                    x[:, -1], y[-1] = xoc.flatten(), yoc
                else:
                    for j in range(1, n+1):
                        x[:, j] = x[:, 0] + 0.5 * (x[:, j] - x[:, 0])
                        y[j] = f(x[:, j])
            else:
                xic = xbar + 0.5 * (x[:, -1].reshape(-1,1) - xbar)
                yic = f(xic.flatten())
                if yic < y[-1]:
                    x[:, -1], y[-1] = xic.flatten(), yic
                else:
                    for j in range(1, n+1):
                        x[:, j] = x[:, 0] + 0.5 * (x[:, j] - x[:, 0])
                        y[j] = f(x[:, j])
    return x, y

# ---------------------------
# Code Block 3: Gradient Descent (Fixed Step)
# ---------------------------
# Copied from professor, no changes

def gradient_descent(
```

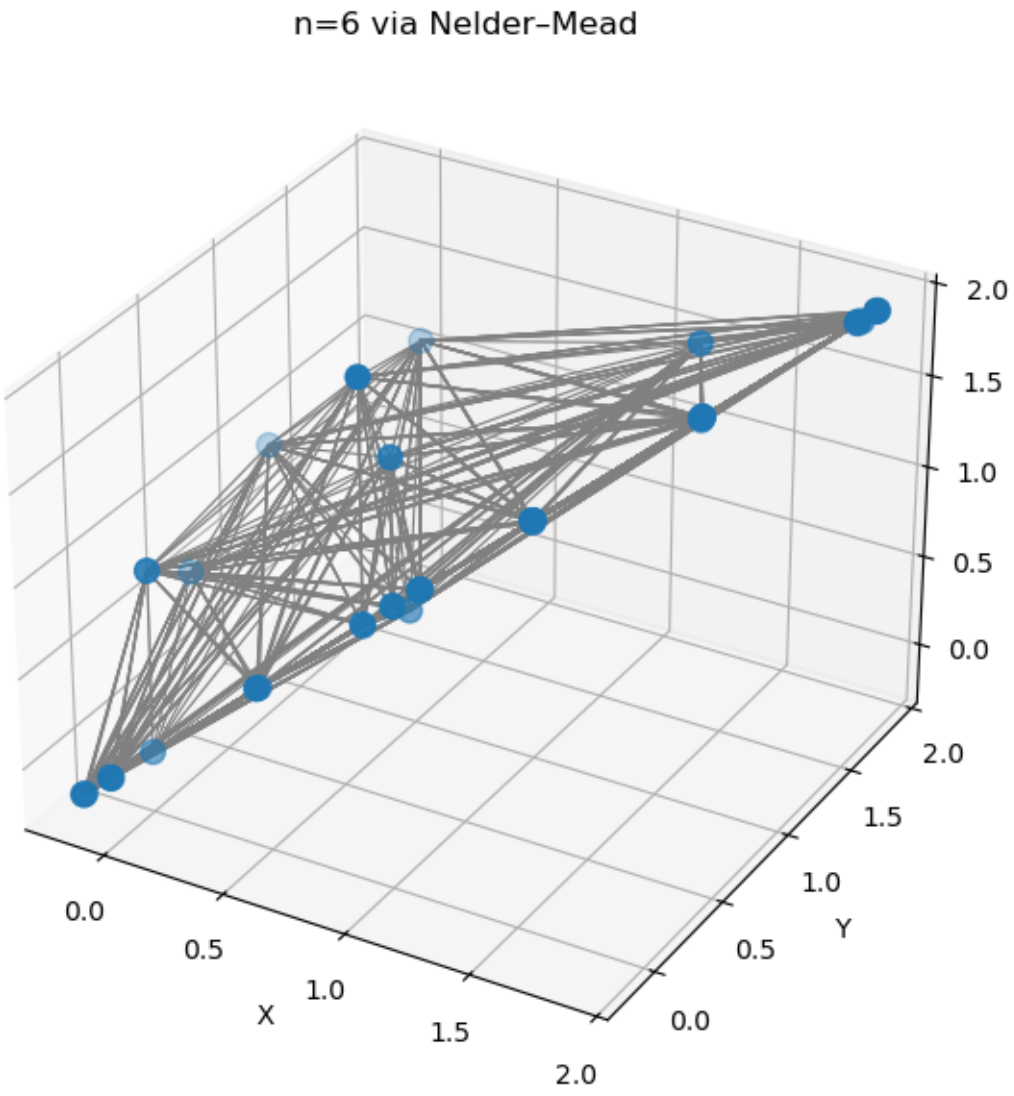
```
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
    eta: float,
    max_iter: int
) -> Tuple[np.ndarray, List[Tuple[np.ndarray, float]]]:
    x = x0.copy(); history = []
    for _ in range(max_iter):
        g = grad_f(x)
        x = x - eta * g
        history.append((x.copy(), f(x)))
    return x, history

# ---------------------------
# Code Block 4: Backtracking GD
# ---------------------------
# Copied from professor, no changes

def gradient_descent_backtracking(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
    eta0: float,
    alpha: float,
    beta: float,
    max_iter: int
) -> Tuple[np.ndarray, List[Tuple[np.ndarray, float]]]:
    x = x0.copy(); history = []
    for _ in range(max_iter):
        g = grad_f(x); t = eta0
        while f(x - t * g) > f(x) - alpha * t * np.dot(g, g):
            t *= beta
        x = x - t * g
        history.append((x.copy(), f(x)))
    return x, history

# ---------------------------
# Code Block 5: Momentum GD
# ---------------------------
# Copied from professor, no changes

def gradient_descent_momentum(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    x0: np.ndarray,
    eta: float,
    mu: float,
```

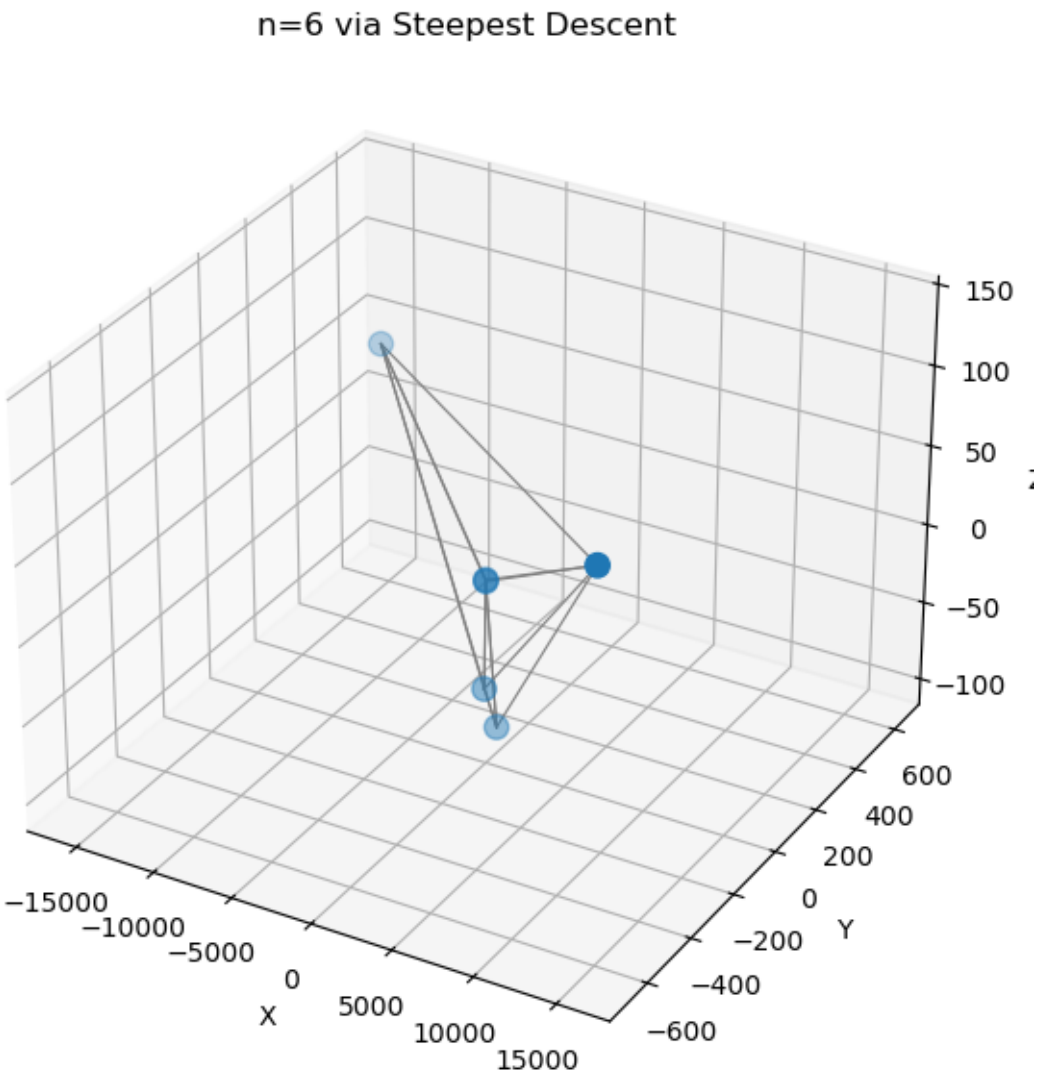
```
    max_iter: int
) -> Tuple[np.ndarray, List[Tuple[np.ndarray, float]]]:
    x = x0.copy(); v = np.zeros_like(x0); history = []
    for _ in range(max_iter):
        g = grad_f(x)
        v = mu * v + eta * g
        x = x - v
        history.append((x.copy(), f(x)))
    return x, history

################################
# Code Block 6: Drivers for n=6
################################
# Random initialization to avoid division-by-zero
n = 6
m = n - 2
x0 = 0.5 + np.random.rand(3 * m)

# Nelder-Mead parameters
dim = x0.size
rad = 0.5
steps = 2000
x_nm6, U_nm6 = nelder_mead(energy, x0, rad, steps)
print("[NM] Min Energy:", np.min(U_nm6))

# Steepest Descent parameters
eta, max_iter = 0.01, 5000
x_gd6, hist_gd6 = gradient_descent(energy, gradient, x0, eta, max_iter)
print("[GD] Min Energy:", energy(x_gd6))

# Backtracking GD parameters
eta0, alpha, beta, max_iter = 0.1, 1e-4, 0.5, 2000
x_bt6, hist_bt6 = gradient_descent_backtracking(energy, gradient, x0, eta0,␣
 ↪alpha, beta, max_iter)
print("[BT] Min Energy:", energy(x_bt6))

# Momentum GD parameters
eta, mu, max_iter = 0.01, 0.9, 2000
x_mom6, hist_mom6 = gradient_descent_momentum(energy, gradient, x0, eta, mu,␣
 ↪max_iter)
print("[MO] Min Energy:", energy(x_mom6))

################################
# Code Block 7: 3D Visualization
################################
from mpl_toolkits.mplot3d import Axes3D  # noqa: F401
import matplotlib.pyplot as plt
```

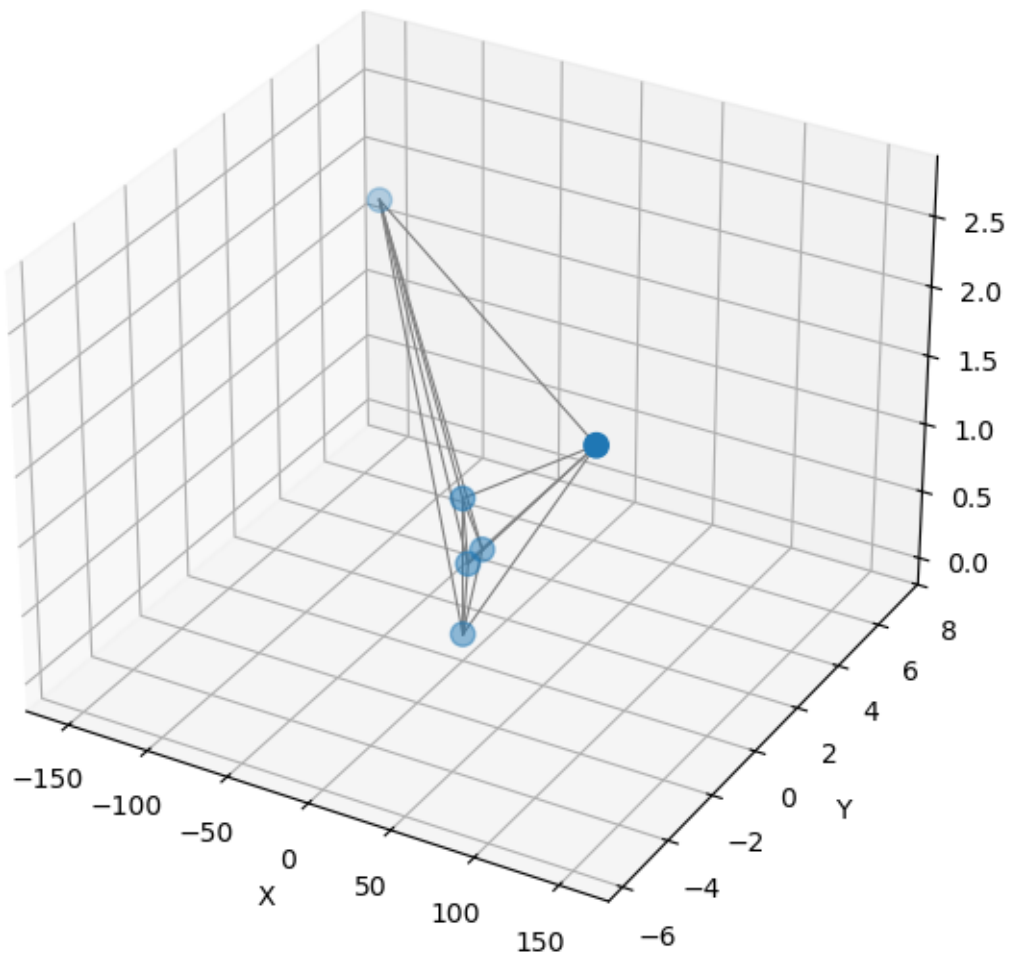
```python
def plot_cluster_3d(free_coords: np.ndarray, title: str = "", save_as: str =␣
 ↪None):
    """
    3D scatter + line plot of an n-atom Lennard-Jones cluster.
    Adapts examples from the Python Data Science Handbook (VanderPlas).
    free_coords: 1D array length 3*(n-2)
    """
    # Rebuild full coords
    m = free_coords.size // 3
    n = m + 2
    coords = np.zeros((n, 3), dtype=float)
    coords[0] = [0, 0, 0]
    coords[1] = [0, 0, 1]
    coords[2:] = free_coords.reshape(m, 3)

    # Unpack for plotting
    xs, ys, zs = coords[:,0], coords[:,1], coords[:,2]

    # Create figure & 3D axes
    fig = plt.figure(figsize=(7,7))
    ax = fig.add_subplot(projection='3d')

    # Scatter atoms
    ax.scatter3D(xs, ys, zs, s=80, c='C0', marker='o', depthshade=True)

    # Draw bonds (all pairwise connections)
    for i in range(n):
        for j in range(i+1, n):
            ax.plot3D([xs[i], xs[j]],
                      [ys[i], ys[j]],
                      [zs[i], zs[j]],
                      color='gray', linewidth=0.8)

    # Labels and title
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title(title)

    if save_as:
        plt.savefig(save_as, dpi=150, bbox_inches='tight')
    plt.show()

# Example calls:
plot_cluster_3d(x_nm6,  title="n=6 via Nelder-Mead")
plot_cluster_3d(x_gd6,  title="n=6 via Steepest Descent")
```

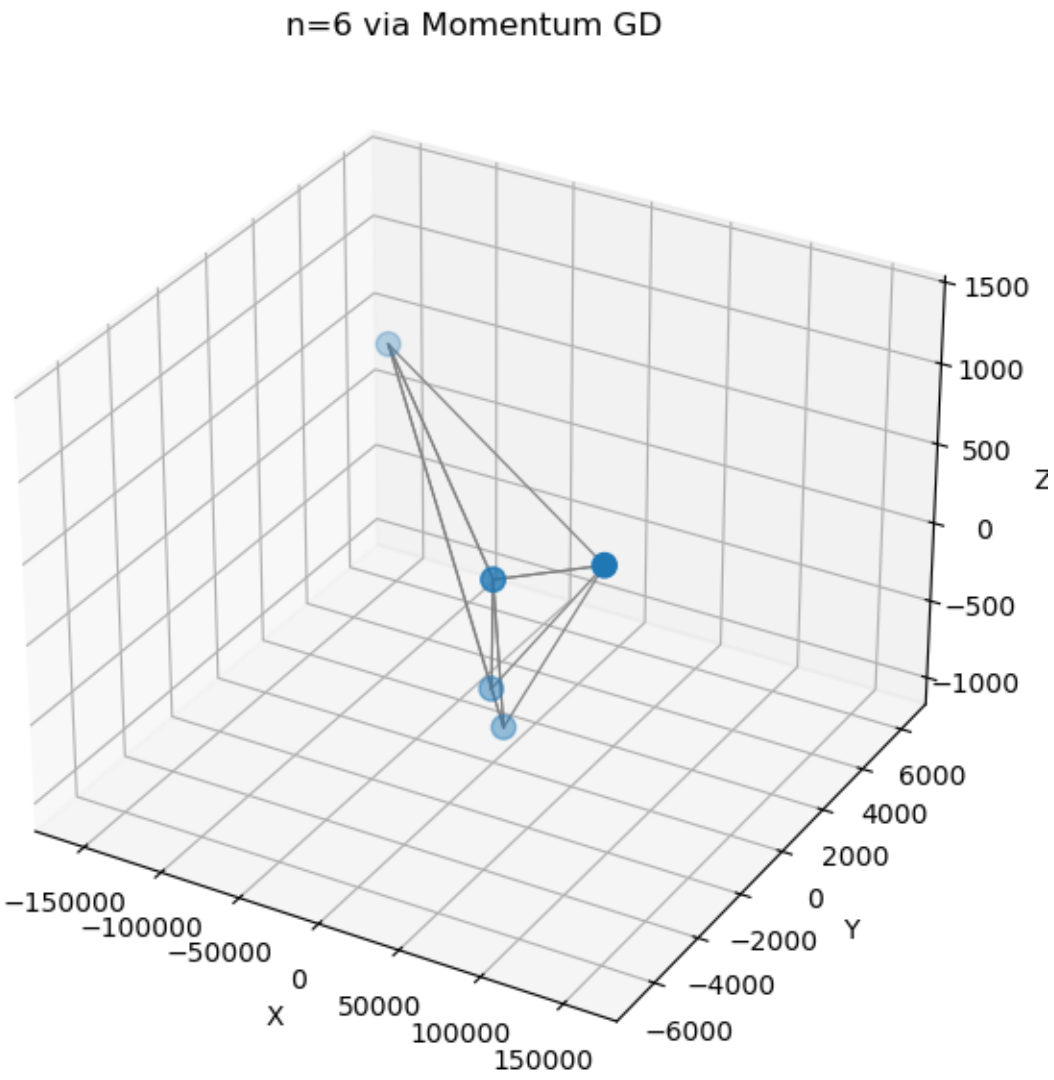
```
plot_cluster_3d(x_bt6,  title="n=6 via Backtracking GD")
plot_cluster_3d(x_mom6, title="n=6 via Momentum GD")
```

```
[NM] Min Energy: -10.291868324491872
[GD] Min Energy: -1.0000000000269702
[BT] Min Energy: -6.0000000000011395
[MO] Min Energy: -1.0
```




n=6 via Steepest Descent
150
100
50
0
−50
−100
600
400
200
0
−200
−400
−600
Y
−15000
−10000
−5000
0
5000
10000
15000
X


n=6 via Backtracking GD
2.5
2.0
1.5
1.0
0.5
0.0
8
6
4
2
0
−2
−4
−6
Y
−150
−100
−50
0
50
100
150
X



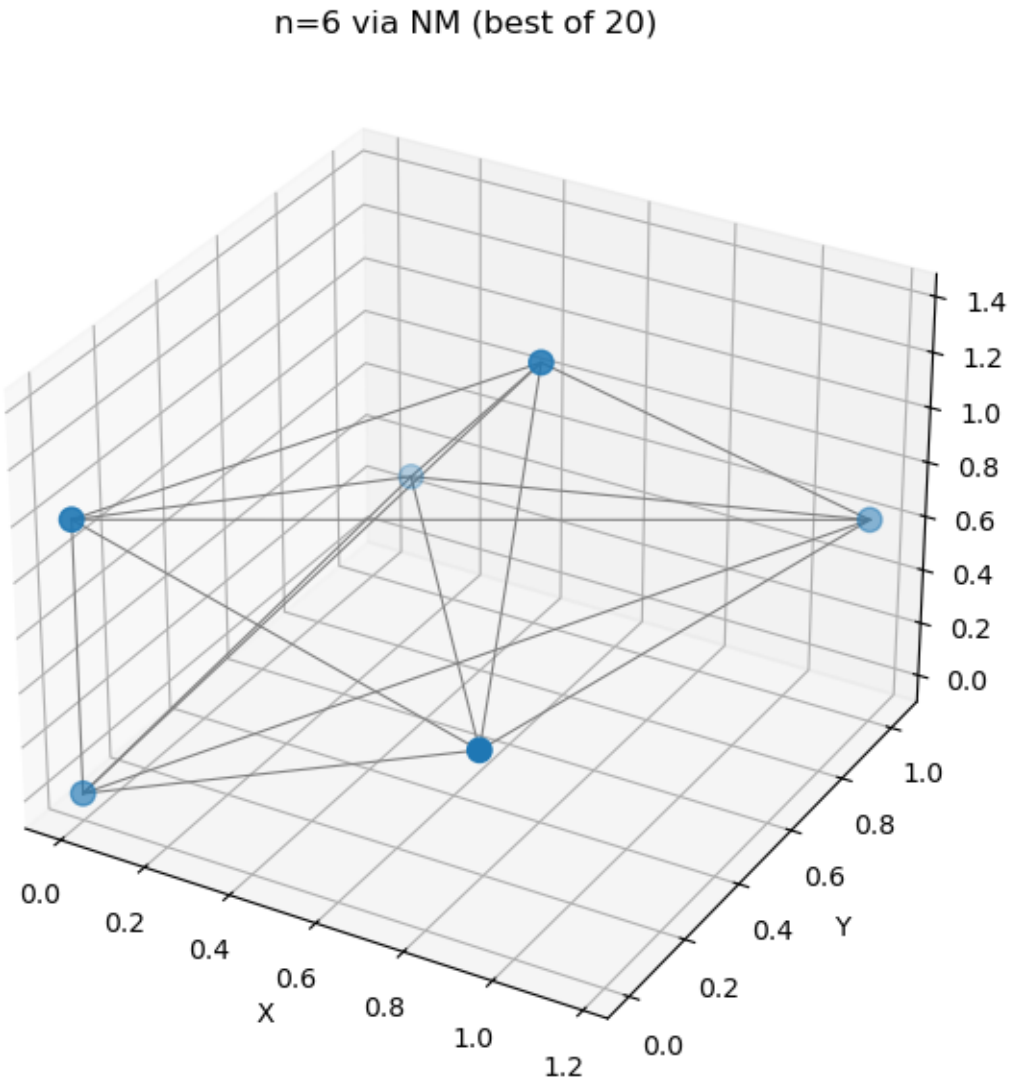
1.6 Multi-start Nelder–Mead for n=6

[53]:
```
# ###Multi-start Nelder-Mead for n=6 ---
best_U6     = np.inf
best_x6     = None
step_counts6 = []

for seed in range(20):
    x0_try = 0.5 + np.random.rand(3*(n-2))
    x_simplex, U_vals = nelder_mead(energy, x0_try, rad, steps)

    U_min      = np.min(U_vals)
    steps_taken = np.argmin(U_vals) + 1
```

```
    step_counts6.append(steps_taken)

    if U_min < best_U6:
        best_U6 = U_min
        best_x6 = x_simplex[:, np.argmin(U_vals)]

print(f"[NM n=6 multi-start] Best energy over 20 runs: {best_U6:.6f}")
print(f"[NM n=6] Steps (min/avg/max): "
      f"{min(step_counts6)}/"
      f"{np.mean(step_counts6):.1f}/"
      f"{max(step_counts6)}")
```

```
[NM n=6 multi-start] Best energy over 20 runs: -12.220872
[NM n=6] Steps (min/avg/max): 1/2.8/13
```

[55]:
```
plot_cluster_3d(best_x, title=f"n={n} via NM (best of 20)")
```



1.7 Multi-start for Gradient Methods on n=6

[57]:
```
###############Multi-start for Gradient Methods on n=6#################
import numpy as np

methods = {
    'GD':   lambda x0: gradient_descent(    energy, gradient, x0,     eta=0.01,␣
 ↪max_iter=5000),
    'BT':   lambda x0: gradient_descent_backtracking(
                           energy, gradient, x0,
                           eta0=0.1, alpha=1e-4, beta=0.5, max_iter=2000),
    'MO':   lambda x0: gradient_descent_momentum(
                           energy, gradient, x0,
                           eta=0.01, mu=0.9, max_iter=2000),
}

n_runs = 20
target6 = best_U6  # from your NM multi-start,   -9.103691

for name, solver in methods.items():
    best_E = np.inf
    step_counts = []
    hits = 0

    for seed in range(n_runs):
        x0_try = 0.5 + np.random.rand(3*(n-2))
        x_final, history = solver(x0_try)
        E_final = energy(x_final)
        steps = len(history)

        step_counts.append(steps)
        if E_final < best_E:
            best_E = E_final
        if abs(E_final - target) < 1e-4:
            hits += 1

    print(f"\n=== {name} ===")
    print(f"Best energy: {best_E:.6f}")
    print(f"Hit global min (±1e-4) in {hits}/{n_runs} runs")
    print(f"Steps: min={min(step_counts)}, avg={np.mean(step_counts):.1f},␣
 ↪max={max(step_counts)}")
```

```
=== GD ===
Best energy: -3.000088
```

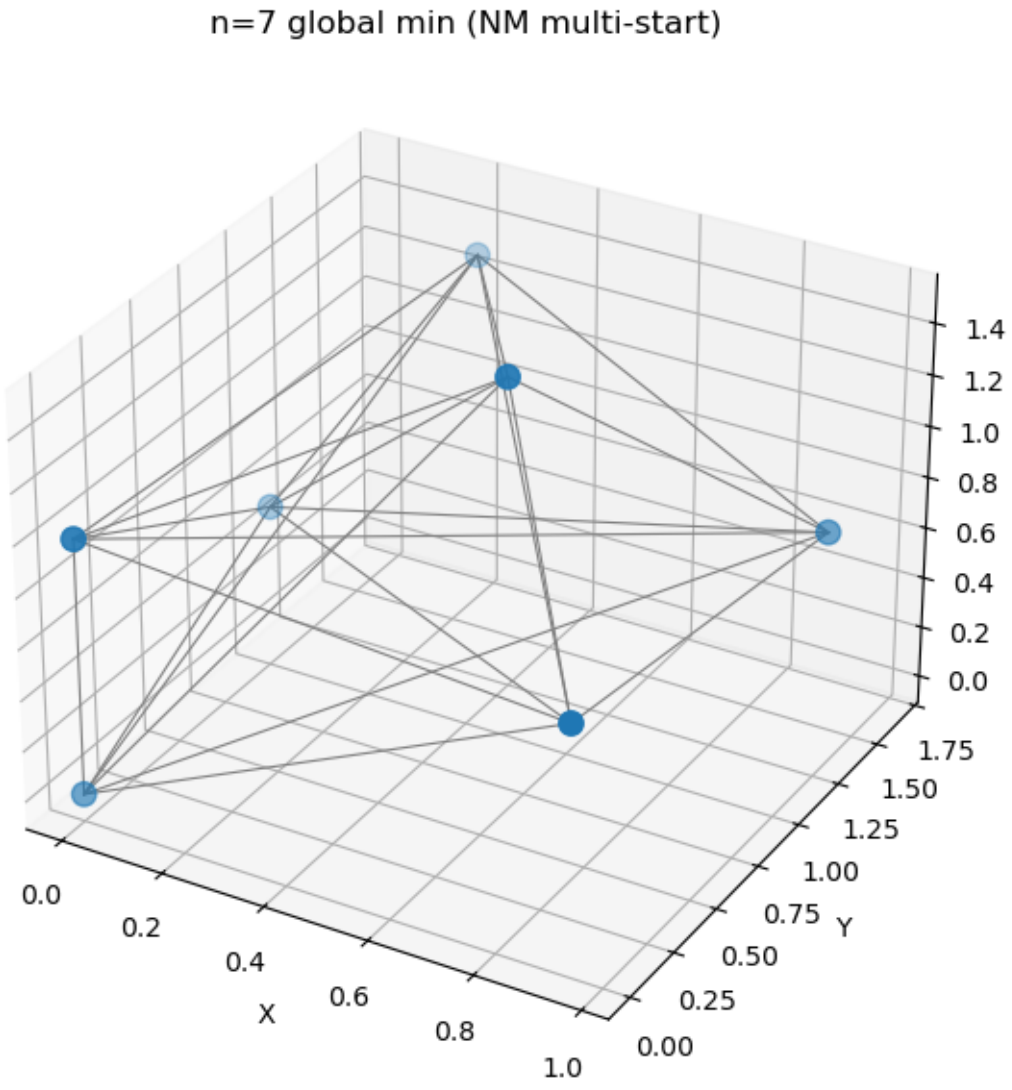
```
Hit global min (±1e-4) in 0/20 runs
Steps: min=5000, avg=5000.0, max=5000

=== BT ===
Best energy: -6.000000
Hit global min (±1e-4) in 0/20 runs
Steps: min=2000, avg=2000.0, max=2000

=== MO ===
Best energy: -1.000441
Hit global min (±1e-4) in 0/20 runs
Steps: min=2000, avg=2000.0, max=2000
```

1.8 End OF CODEBLOCK FOR N=6

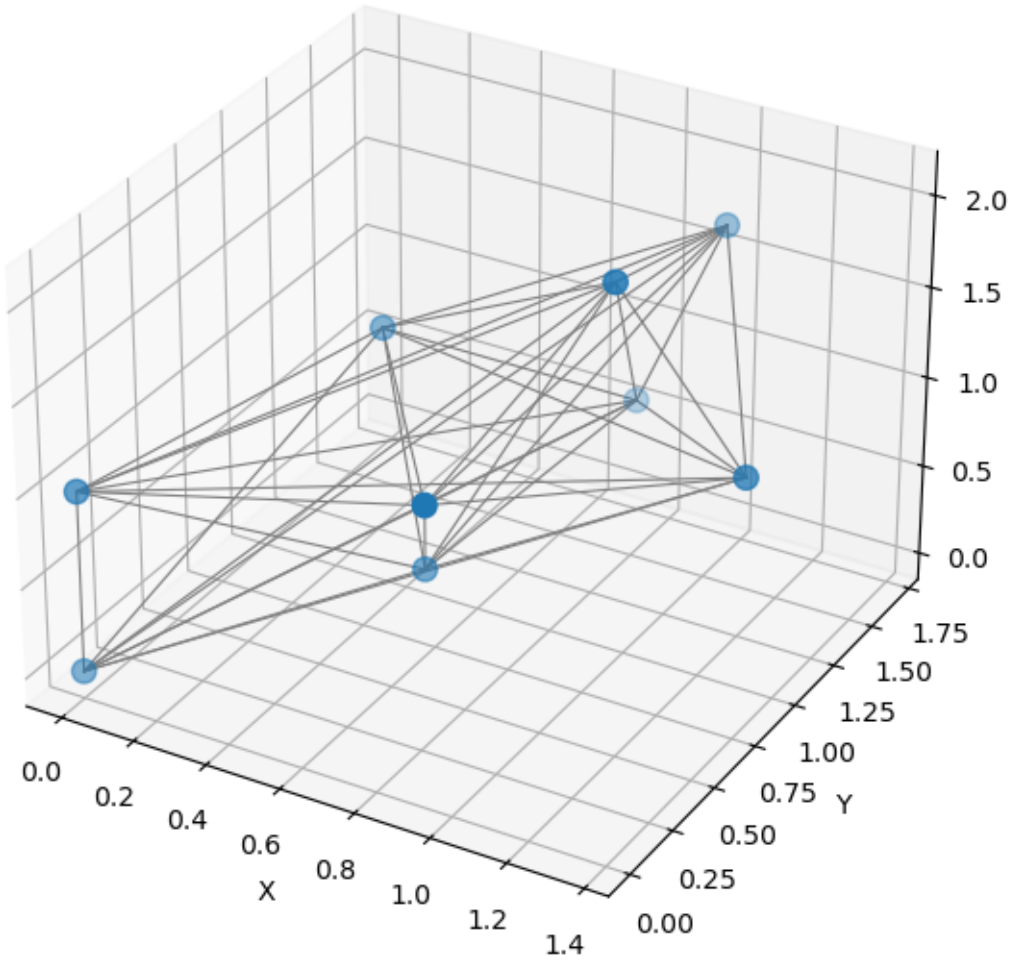
1.9 Larger-n Experiments: in NELDER-MEAD SOLVER ———

[59]:
```python
# --- Larger-n Experiments: in NELDER-MEAD SOLVER ---
import numpy as np

n_list = [7, 9, 11]
rad, steps = 0.5, 2000
results = []

for n in n_list:
    m = n - 2
    best_Un = np.inf
    best_xn = None

    for seed in range(10):   # 10 starts to save time - I was timing out
        x0_try = 0.5 + np.random.rand(3*m)
        x_s, U_vals = nelder_mead(energy, x0_try, rad, steps)
        U_min = np.min(U_vals)
        if U_min < best_Un:
            best_Un = U_min
            best_xn = x_s[:, np.argmin(U_vals)]

    print(f"\nn={n}: best energy  {best_Un:.6f}")
    plot_cluster_3d(best_xn, title=f"n={n} global min (NM multi-start)")
    results.append((n, best_Un))

# for report - tabulate results
print("\nSummary of larger-n energies:")
for n, U in results:
    print(f" n={n}:  U  {U:.6f}")
```

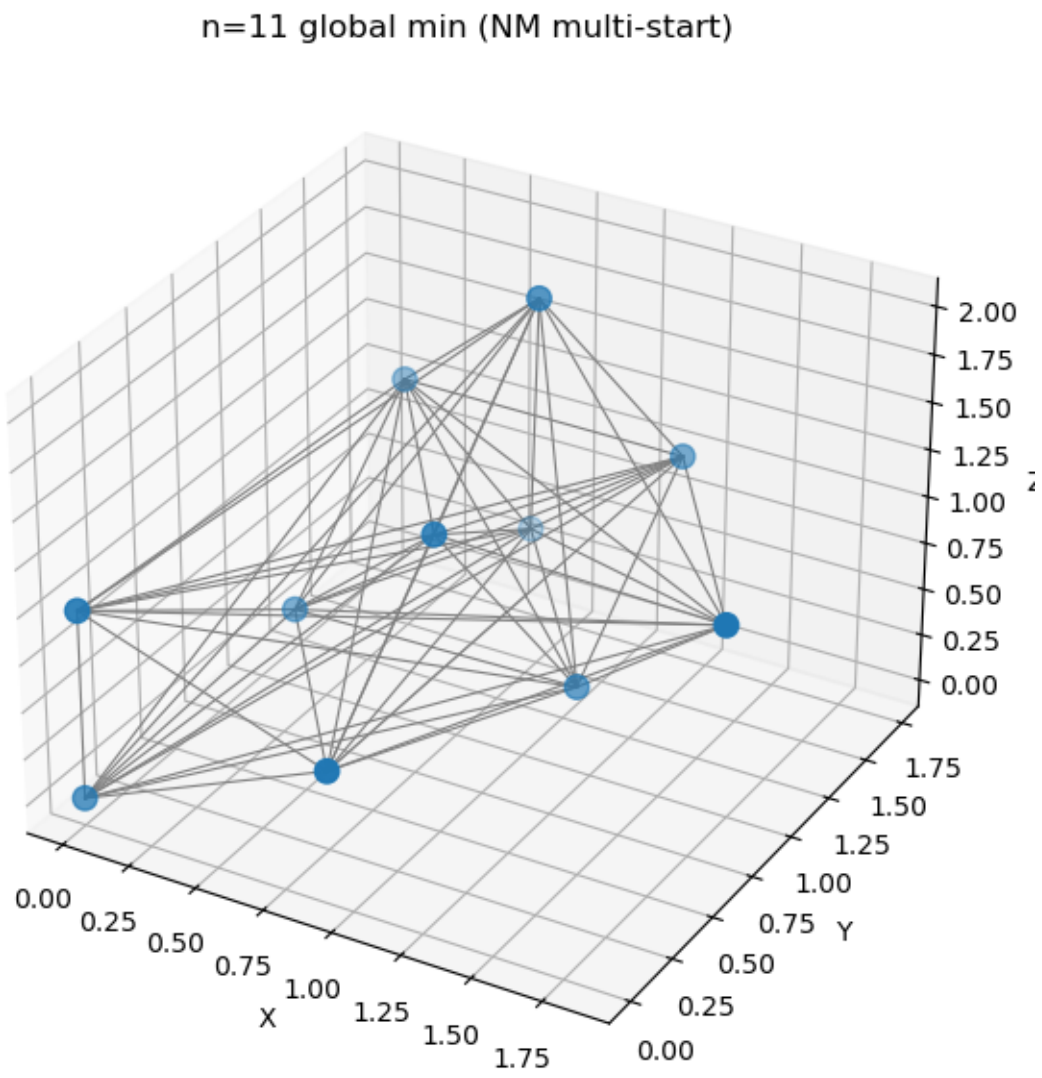
```
n=7: best energy  -14.635853
```




```
n=9: best energy   -21.760120
```




```
n=11: best energy  -29.202434
```




```
Summary of larger-n energies:
 n=7:  U   -14.635853
 n=9:  U   -21.760120
 n=11:  U   -29.202434
```

1.10 End Of Code