Mechanosensitive Behaviour of Collective Cells Modelled as Semi-Flexible Ring Polymers and Inverse Modelling using ML

Durgesh Kumar Patel (21MS109)

Supervisor: Prof. Rumi De
Department of Physical Sciences

Co-Supervisor: Prof. Monidipa Das
Department of Computational and Data Sciences

MS Thesis Report
December 9, 2025

Department of Physical Sciences
Indian Institute of Science Education and Research, Kolkata

Abstract

The mechanical rigidity and deformability of cells are fundamental drivers of collective biological phenomena, ranging from tissue morphogenesis to the metastatic progression of cancer cells. We modelled collective cells as disjoint, semi-flexible ring polymers via coarse-grained Langevin dynamics to investigate mechanosensitive behaviour, and subsequently employed geometric deep learning for inverse modelling to infer intrinsic rigidity from cellular shapes. Our simulations reveal density-driven conformational transitions, where increasing confinement and bending stiffness shift cellular morphologies from isotropic, rough geometries to highly anisotropic, biconcave structures. Addressing the challenge of non-invasively quantifying these properties, we benchmarked a Convolutional Neural Network (CNN) trained directly on simulation images against a novel Graph Neural Network (GNN) architecture. While the image-based CNN yielded moderate success, the GNN—which explicitly encodes topological connectivity—achieved superior accuracy, demonstrating that graph-based machine learning can effectively decode the physical state of soft matter systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Biological Relevance

Cells constitute the fundamental units of living organisms, and their ability to move is a critical aspect of their function. While unicellular organisms often utilize simple structures like flagella, cell movement in multicellular organisms involves complex mechanisms driven by cytoskeletal rearrangement and interactions with the extracellular matrix (ECM) . This phenomenon, known as cell migration, is essential for physiological processes such as embryogenesis, wound healing, and immune system function.

However, cell migration plays a detrimental role in the context of cancer metastasis. Metastasis is the spread of cancer from a primary site to distant organs and is responsible for the majority of cancer-related mortalities [1]. The invasion-metastasis cascade involves dissemination from the primary tumor, intravasation into the bloodstream, and extravasation into secondary tissues.

Recently, increasing evidence suggests that mechanical cues—specifically cell rigidity and deformability—play a pivotal role in this progression alongside biochemical signals. Physical plasticity allows cells to navigate through the dense ECM and narrow endothelial junctions. Experimental studies using atomic force microscopy have demonstrated that metastatic cancer cells can be up to 70% softer than their non-metastatic counterparts [2]. Furthermore, microfluidic studies indicate that increased deformability directly correlates with higher invasive potential [3].

Therefore, understanding how single-cell mechanics (stiffness) and collective behavior (density) influence conformational changes is crucial for elucidating the physical mechanisms of metastasis.

1.2 Why this Model?

To investigate these mechanosensitive behaviors, we model the cells as semi-flexible ring polymers. This coarse-grained (CG) approach is motivated by several biological and physical factors:

1. **Biological Analogies:** The ring polymer topology characterizes various biological systems, such as circular DNA in viral genomes and bacterial plasmids [8]. Furthermore, in the context of cellular mechanics, the ring polymer serves as a minimal model for the cell cortex—a semi-flexible actin network constrained in a closed loop.

2. **2D Confinement:** We consider cells fully adsorbed on a substrate, effectively treating the system in two spatial dimensions. This setup mimics in vitro experiments of cell migration on flat surfaces and DNA adsorbed onto lipid membranes [8].

3. **Conformational Richness:** As established by Zhu et al. [8], disjoint semi-flexible ring polymers exhibit rich phase behavior driven by the interplay between bending stiffness (κ) and areal density (ρ). At low densities, thermal fluctuations drive the system. However, at densities above the overlap concentration (ρ^*), crowding effects induce shape transitions from circular to obround and biconcave geometries [8].

This model captures the essential physics of "crowding-induced" deformation. By linking the bending stiffness (representing cell cortex rigidity) and density to observed conformational changes, we can simulate the transition from healthy (rigid/circular) to invasive (soft/anisotropic) phenotypes seen in metastatic progression.

1.3 Inverse Modelling and Machine Learning

While Molecular Dynamics (MD) simulations allow us to predict cell shapes based on known mechanical parameters (Forward Modelling), the practical biological challenge is often the reverse: inferring mechanical properties (e.g., stiffness) from static microscopic images of cell shapes. This is known as Inverse Modelling.

The relationships between collective cell architecture, individual shapes, and underlying interaction parameters are high-dimensional and non-linear. Traditional analytical inversion is often intractable for such complex many-body systems. Machine Learning (ML) has emerged as a powerful tool to bridge this gap [6, 5].

- **Surrogate Modelling:** ML algorithms can serve as efficient surrogates for computationally expensive MD simulations. By training on dataset generated from coarse-grained simulations, ML models can learn the statistical mapping between sequence/structure and function [6, 5].

- **Mechanical Inference:** In this work, we aim to develop an ML framework that utilizes "images" of cell configurations (simulation snapshots) to predict the underlying bending rigidity (κ). This has significant diagnostic relevance; if a model can accurately predict stiffness from shape, it could potentially identify invasive metastatic cells solely based on visual data from biopsies, without invasive mechanical testing.

- **Design and Discovery:** Beyond prediction, integrating ML with CGMD allows for the exploration of the "polymer genome," enabling the inverse design of materials or cellular systems with targeted structural properties [6].

Chapter 2

Model

In this chapter, we discuss the details of the spring-bead model used to simulate the collective behavior of cells. We define the equations of motion governing the system and derive the specific forces acting on the monomers from their respective potential energy functions.

2.1 Spring Bead Model

We utilize a coarse-grained spring-bead model where cells are conceptualized as non-overlapping ring polymers [8]. Each ring consists of N beads connected by linear springs, capturing the essential biological features of flexibility, elasticity, and topological constraints. This approach is similar to models used by Mkrtchyan et al. [4] and Wen et al. [7] to study tissue topology and collective cell motion.

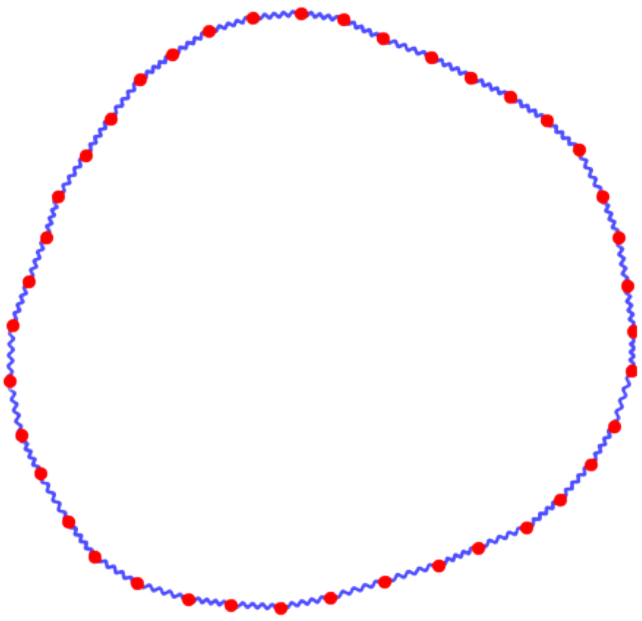


Figure 2.1: Cartoon of the spring bead model. Orange circles represent monomers connected by springs.

2.2 Dynamics: Langevin Equation

The dynamics of each bead i are governed by the Langevin equation, which describes the motion of a particle in a fluid (implicit solvent) subject to fluctuating thermal forces [8]:

$$\dot{\mathbf{r}}_i = \mathbf{v}_i \tag{2.1}$$

$$m\dot{\mathbf{v}}_i = -\nabla_i U_{total} - \Gamma \mathbf{v}_i + \boldsymbol{\Xi}_i \tag{2.2}$$

where $\mathbf{r}_i$ and $\mathbf{v}_i$ are the position and velocity vectors, m is the mass, Γ is the friction coefficient, and $\boldsymbol{\Xi}_i$ is the Gaussian white noise satisfying the fluctuation-dissipation theorem:

$$\langle \xi_{\mu,i}(t)\xi_{\nu,j}(t')\rangle = \sqrt{2\Gamma k_B T}\delta_{\mu\nu}\delta_{ij}\delta(t-t') \tag{2.3}$$

2.3 Force Definitions and Derivations

The total force on a monomer is derived from the gradient of the total potential energy U_{total}. In this section, we derive the expressions for the Spring, Bending, and Repulsion forces.

2.3.1 Spring Force

The neighboring monomers in the ring are connected by harmonic springs to maintain structural integrity.

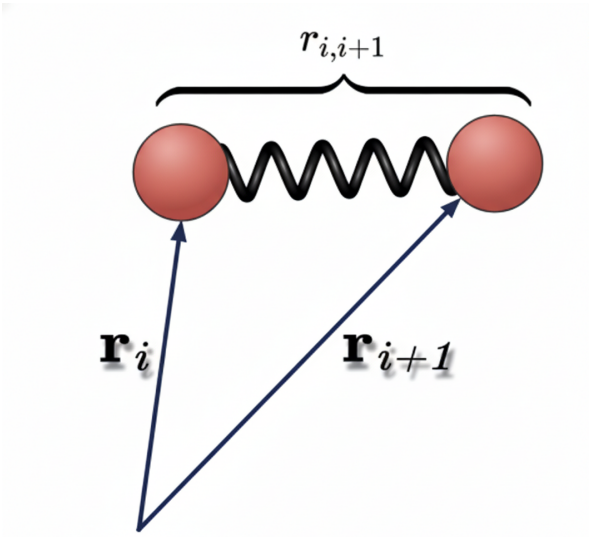


Figure 2.2: Vectors involved in the spring potential between beads i and $i+1$.

Derivation

The spring potential between bead i and its neighbor $i+1$ is given by:

$$U_{spring}(r_i, r_{i+1}) = \frac{k}{2}(r_{i,i+1} - r_b)^2 \tag{2.4}$$

where $r_{i,i+1} = |\mathbf{r}_{i+1} - \mathbf{r}_i|$ is the distance between beads, r_b is the equilibrium bond length, and k is the spring constant.

The force on bead i in the α-direction ($\alpha = x, y$) is the negative gradient of the potential:

$$F_{spring,\alpha_i} = -\frac{\partial U_{spring}}{\partial \alpha_i} = -k(r_{i,i+1} - r_b)\frac{\partial r_{i,i+1}}{\partial \alpha_i} \tag{2.5}$$

We compute the derivative of the distance $r_{i,i+1} = \sqrt{(\alpha_{i+1} - \alpha_i)^2 + \ldots}$:

$$\frac{\partial r_{i,i+1}}{\partial \alpha_i} = \frac{2(\alpha_{i+1} - \alpha_i)(-1)}{2r_{i,i+1}} = -\frac{\alpha_{i+1} - \alpha_i}{r_{i,i+1}} \tag{2.6}$$

Substituting this back into Eq. (2.5):

$$\begin{aligned} F_{spring,\alpha_i} &= k(r_{i,i+1} - r_b)\left(\frac{\alpha_{i+1} - \alpha_i}{r_{i,i+1}}\right) \\ &= k\left(1 - \frac{r_b}{r_{i,i+1}}\right)(\alpha_{i+1} - \alpha_i) \end{aligned} \tag{2.7}$$

Considering both neighbors ($i-1$ and $i+1$), the total spring force on bead i is:

$$\mathbf{F}_{spring,i} = k\left[\left(1 - \frac{r_b}{r_{i,i+1}}\right)(\mathbf{r}_{i+1} - \mathbf{r}_i) + \left(1 - \frac{r_b}{r_{i,i-1}}\right)(\mathbf{r}_{i-1} - \mathbf{r}_i)\right] \tag{2.8}$$

2.3.2 Bending Force

To model the semi-flexibility of the cell cortex, we introduce a three-body bending potential dependent on the angle θ_i formed by the triplet $(i-1, i, i+1)$.

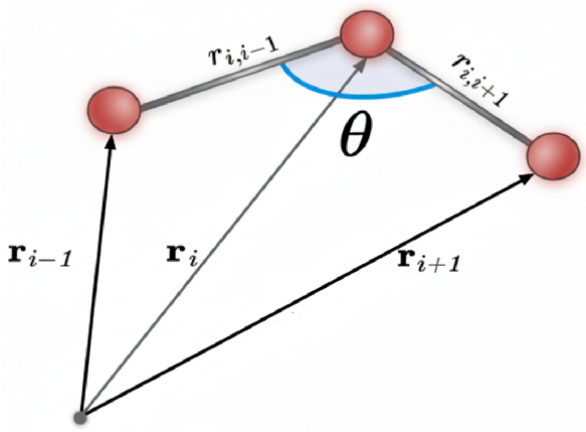


Figure 2.3: Vectors involved in the bending potential defined by angle θ.

Derivation

The potential is defined as:

$$U_{bend} = \kappa(\cos\theta_i - 1) \quad \text{where} \quad \cos\theta_i = \frac{\mathbf{r}_{i-1,i} \cdot \mathbf{r}_{i+1,i}}{|\mathbf{r}_{i-1,i}||\mathbf{r}_{i+1,i}|} \tag{2.9}$$

The total bending force on particle i arises from three contributions: the triplet centered at i, and the triplets centered at $i-1$ and $i+1$.

$$\mathbf{F}_{bend,i} = -\nabla_{\mathbf{r}_i}\left[U^{(i-1,i,i+1)} + U^{(i-2,i-1,i)} + U^{(i,i+1,i+2)}\right] \tag{2.10}$$

Part 1: Force from Main Triplet $(i-1, i, i+1)$
Let $\mathbf{a} = \mathbf{r}_{i-1} - \mathbf{r}_i$ and $\mathbf{b} = \mathbf{r}_{i+1} - \mathbf{r}_i$. Let $A = |\mathbf{a}|$ and $B = |\mathbf{b}|$. The gradient of $\frac{\mathbf{a}\cdot\mathbf{b}}{AB}$ involves the quotient rule. Note that $\nabla_{\mathbf{r}_i}(\mathbf{a}\cdot\mathbf{b}) = -(\mathbf{a}+\mathbf{b})$. The resulting force contribution is:

$$\mathbf{F}_i^{(i-1,i,i+1)} = -\kappa\left[-\frac{\mathbf{a}+\mathbf{b}}{AB} + \frac{(\mathbf{a}\cdot\mathbf{b})\mathbf{a}}{A^3B} + \frac{(\mathbf{a}\cdot\mathbf{b})\mathbf{b}}{AB^3}\right] \tag{2.11}$$

This corresponds to the first term in the final force equation.

Part 2: Force from Left Triplet $(i-2, i-1, i)$
Let $\mathbf{a} = \mathbf{r}_{i-2} - \mathbf{r}_{i-1}$ and $\mathbf{b} = \mathbf{r}_i - \mathbf{r}_{i-1}$. Here, $\nabla_{\mathbf{r}_i}$ only affects vector $\mathbf{b}$. Using $\nabla_{\mathbf{r}_i}\mathbf{b} = \mathbf{I}$, the force simplifies to:

$$\mathbf{F}_i^{(i-2,i-1,i)} = -\frac{\kappa}{A}\left(\frac{\mathbf{a}}{B} - \frac{(\mathbf{a}\cdot\mathbf{b})\mathbf{b}}{B^3}\right) \tag{2.12}$$

Reverting to position vector notation ($\mathbf{r}_{ji}$), this matches the second term (Eq. 2.10).

Part 3: Force from Right Triplet $(i, i+1, i+2)$
By symmetry with Part 2, replacing indices $(i-2 \to i, i-1 \to i+1, i \to i+2)$, we obtain the third term (Eq. 2.10).

Total Bending Force: Summing these contributions yields the full expression implemented in the simulation:

$$\mathbf{F}_{bend,i} = -\kappa\frac{\mathbf{r}_{i-1,i}\cdot\mathbf{r}_{i+1,i}}{r_{i-1,i}r_{i+1,i}}(\dots) - \kappa\frac{\mathbf{r}_{i-2,i-1}\cdot\mathbf{r}_{i,i-1}}{r_{i-2,i-1}r_{i,i-1}}(\dots) - \kappa\frac{\mathbf{r}_{i+2,i+1}\cdot\mathbf{r}_{i,i+1}}{r_{i+2,i+1}r_{i,i+1}}(\dots) \tag{2.13}$$

2.3.3 Repulsion Force

To prevent self-intersection and overlap between cells (volume exclusion), we apply a soft repulsive potential between non-bonded beads.

Derivation

The potential is defined as a quadratic penalty that activates only when the distance r_{ij} is less than a cutoff r_m:

$$U_{rep}(r_{ij}) = \begin{cases} \frac{\chi}{2}\left(1 - \frac{r_{ij}}{r_m}\right)^2 & \text{if } r_{ij} \le r_m \\ 0 & \text{if } r_{ij} > r_m \end{cases} \tag{2.14}$$

The force on particle i is $\mathbf{F}_i = -\nabla_{\mathbf{r}_i}U_{rep}$. Using the chain rule:

$$\mathbf{F}_i = -\frac{dU_{rep}}{dr_{ij}}\nabla_{\mathbf{r}_i}r_{ij} \tag{2.15}$$

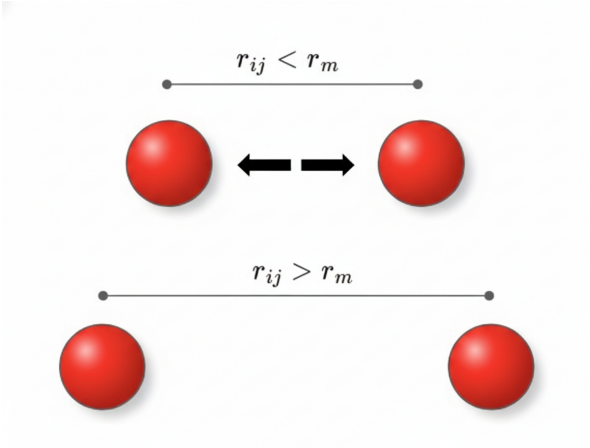


Figure 2.4: Repulsion potential within the cutoff r_m.

First, the derivative of the potential with respect to distance:

$$\frac{dU_{rep}}{dr_{ij}} = \frac{\chi}{2} \cdot 2\left(1 - \frac{r_{ij}}{r_m}\right) \cdot \left(-\frac{1}{r_m}\right) = -\frac{\chi}{r_m}\left(1 - \frac{r_{ij}}{r_m}\right) \tag{2.16}$$

Second, the gradient of the distance $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$ with respect to $\mathbf{r}_i$:

$$\nabla_{\mathbf{r}_i} r_{ij} = -\frac{\mathbf{r}_j - \mathbf{r}_i}{r_{ij}} = -\frac{\mathbf{r}_{ji}}{r_{ij}} \tag{2.17}$$

Multiplying these terms:

$$\begin{aligned} \mathbf{F}_i &= -\left[-\frac{\chi}{r_m}\left(1 - \frac{r_{ij}}{r_m}\right)\right]\left[-\frac{\mathbf{r}_{ji}}{r_{ij}}\right] \\ &= -\frac{\chi}{r_m}\left(\frac{1}{r_{ij}} - \frac{1}{r_m}\right) r_{ij} \cdot \frac{\mathbf{r}_{ji}}{r_{ij}} \\ &= \frac{\chi}{r_m}\left(\frac{1}{r_m} - \frac{1}{r_{ij}}\right)(\mathbf{r}_j - \mathbf{r}_i) \end{aligned} \tag{2.18}$$

This yields the final expression used in the model:

$$F_{rep,\alpha_i} = \frac{\chi}{r_m} \sum_{j \neq neighbors} \left(\frac{1}{r_m} - \frac{1}{r_{ij}}\right)(\alpha_j - \alpha_i) \tag{2.19}$$

Chapter 3

Simulation Setup

In this chapter, we detail the computational framework established to simulate the collective behavior of the cells. We describe the system configuration, the initialization protocols, and the specific numerical integration scheme employed to evolve the system over time.

3.1 System Configuration

The simulation domain consists of a two-dimensional square box with periodic boundary conditions (PBC) applied in both x and y directions to eliminate edge effects and mimic a bulk cellular environment.

The key system parameters are defined as follows:

- **Cell Representation:** Each cell is modeled as a ring polymer composed of $N = 40$ beads connected by harmonic springs in a circular loop topology.
- **Total System Size:** The full system comprises $N_{cells} = 36$ cells, resulting in a total of 1440 beads.
- **Confinement:** The periodic boundary conditions ensure that a cell exiting one side of the simulation box re-enters from the opposite side, maintaining a constant number of particles and density within the simulation volume. (Figure 3.1)

3.2 Initialization

To ensure an unbiased starting configuration, the system is initialized using a random distribution protocol (Figure 3.2). The initialization steps are as follows:

1. **Center Placement:** The centers of the 36 cells are selected randomly within the simulation box domain.
2. **Bead Distribution:** For each cell, the 40 beads are distributed in a perfect circle around the chosen center. The radius of this initial circle is set such that the bond lengths are close to the equilibrium length r_b.
3. **Velocity Initialization:** The initial velocity of every bead is set to zero ($v_i = 0$). The system is then allowed to thermalize, gaining kinetic energy from the stochastic noise term in the Langevin equation.

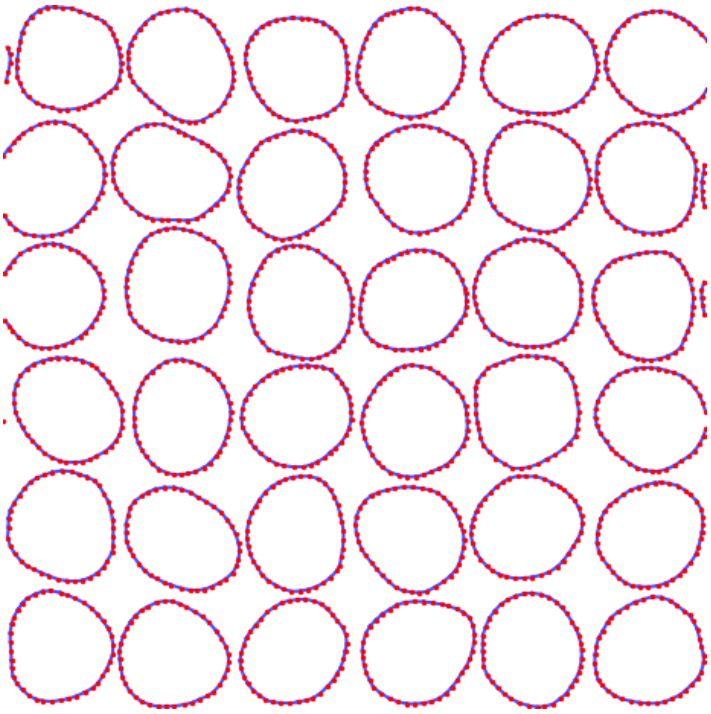


Figure 3.1: System configuration showing 36 cells with 40 beads each in a periodic box.

3.3 Numerical Integration: Modified Velocity-Verlet

The equations of motion are integrated using a modified Velocity-Verlet algorithm adapted for Langevin dynamics. This semi-implicit scheme ensures stability and accuracy in the presence of the friction and random force terms.

The integration proceeds in the following steps:

1. **Position Update:** Calculate the new position $x(t+dt)$ using the current velocity and force:

$$x(t+dt) = x(t) + bv(t)dt + \frac{bF(t)}{2m}dt^2 + \frac{b\Xi(t+dt)}{2m}dt \tag{3.1}$$

2. **Force Calculation:** Calculate the new force $F(t+dt)$ based on the updated positions $x(t+dt)$.

3. **Velocity Update:** Calculate the new velocity $v(t+dt)$ using both the old and new forces:

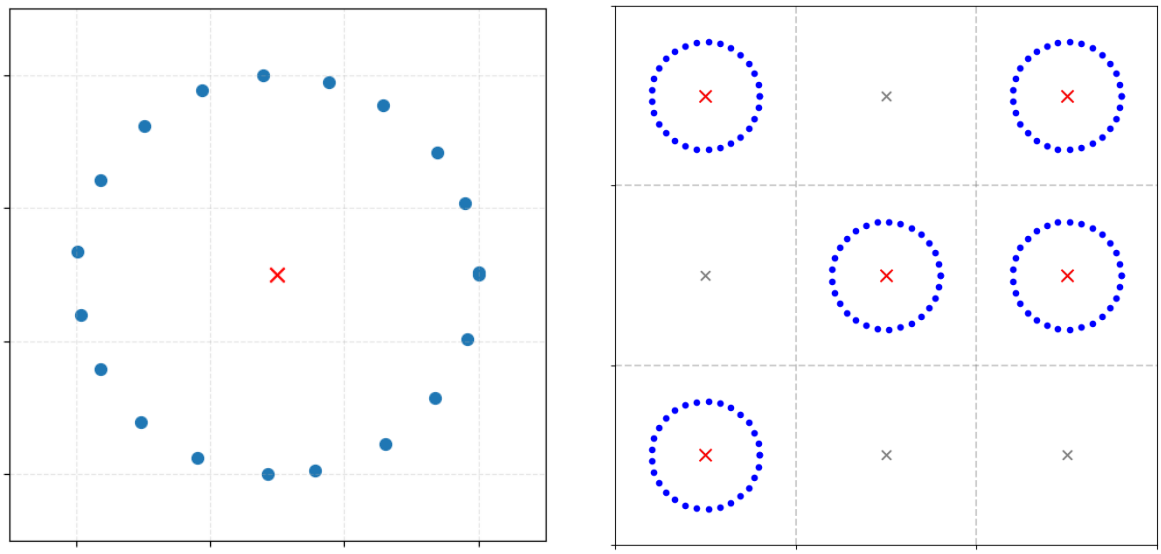
$$v(t+dt) = av(t) + \frac{dt}{2m}(aF(t) + F(t+dt)) + \frac{b\Xi(t+dt)}{m} \tag{3.2}$$

The coefficients a and b incorporate the friction coefficient Γ:

$$a = \frac{1 - \frac{\Gamma dt}{2m}}{1 + \frac{\Gamma dt}{2m}}, \quad b = \frac{1}{1 + \frac{\Gamma dt}{2m}} \tag{3.3}$$

3.4 Simulation Parameters

The following parameters were held constant for the production runs:(Table 3.1)



(a) Initialization grid.

(b) Randomly selected cell centers.

Figure 3.2: Initialization process.

Parameter	**Value**
Number of beads per cell (N)	40
Spring Constant (k)	$100.0\, k_B T / r_b^2$
Time Step (dt)	$1 \times 10^{-3} \tau$
Total Steps	15,000
Repulsion Strength (χ)	$100.0\, k_B T$
Friction Coefficient (Γ)	$0.5\, m/\tau$
Noise Strength (η)	0.002

Table 3.1: System Parameter Values used in the simulation.

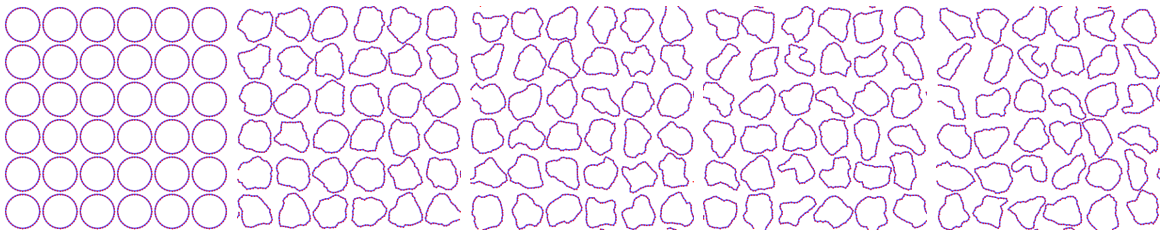
Chapter 4

Simulation Results

In this chapter, we present the visual outcomes of the molecular dynamics simulations. We first illustrate the temporal evolution of the system through snapshots taken from the simulation videos for select parameter sets. Subsequently, we provide a comprehensive matrix of steady-state configurations covering the full range of explored densities (ρ) and bending rigidities (κ).

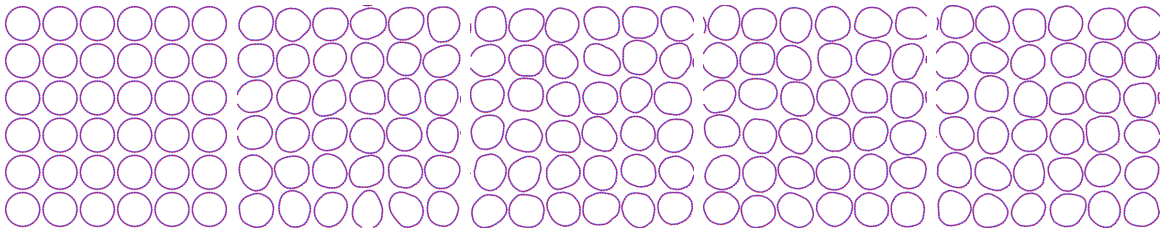
4.1 Temporal Evolution of Cell Conformations

To visualize the dynamics of the system, we present time-series snapshots for four distinct simulation cases representing the extremes of our parameter space: low vs. high density and flexible vs. rigid cells.



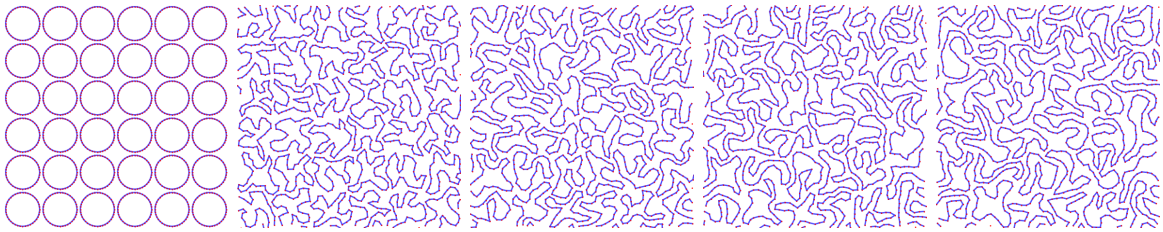
(a) Step: 0 (b) Step: 3750 (c) Step: 7500 (d) Step: 11250 (e) Step: 14999

Figure 4.1: Temporal evolution for Low Density ($\rho \approx 0.0051 r_b^{-2}$) and Bending Stiffness ($\kappa = 5$). Flexible cells fluctuate significantly due to thermal noise without strong shape constraints.



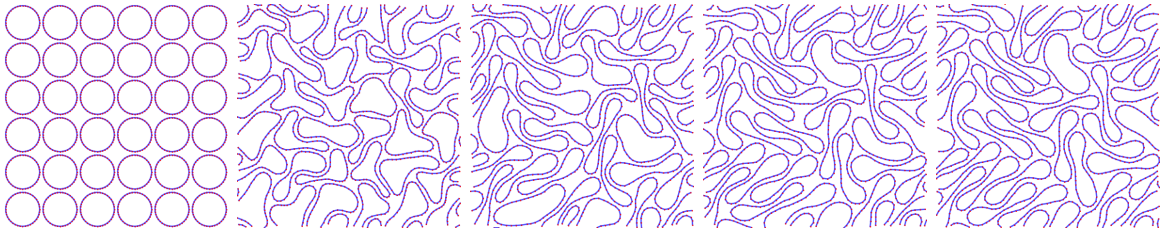
(a) Step: 0 (b) Step: 3750 (c) Step: 7500 (d) Step: 11250 (e) Step: 14999

Figure 4.2: Temporal evolution for Low Density ($\rho \approx 0.0051 r_b^{-2}$) and High Stiffness ($\kappa = 100$). Cells maintain a circular profile due to high bending energy.



(a) Step: 0 (b) Step: 3750 (c) Step: 7500 (d) Step: 11250 (e) Step: 14999

Figure 4.3: Temporal evolution for High Density ($\rho \approx 0.0152 r_b^{-2}$) and Bending Stiffness ($\kappa = 5$). Crowding effects force flexible cells into highly irregular, interdigitated shapes.



(a) Step: 0 (b) Step: 3750 (c) Step: 7500 (d) Step: 11250 (e) Step: 14999

Figure 4.4: Temporal evolution for High Density ($\rho \approx 0.0152 r_b^{-2}$) and High Stiffness ($\kappa = 100$). High rigidity resists deformation, but crowding forces a transition to ordered, anisotropic packing.

4.2 Steady-State Conformations: Full Parameter Sweep

We performed a systematic parameter sweep to map the conformational phase space. The following matrix displays the steady-state snapshots ($t = 15,000$) for all combinations of bending stiffness $\kappa \in [0, 2.5, 5, 10, 20, 40, 100]$ and areal density $\rho \in [0.0051, 0.0076, 0.0101, 0.0126, 0.0152]\, r_b^{-2}$.

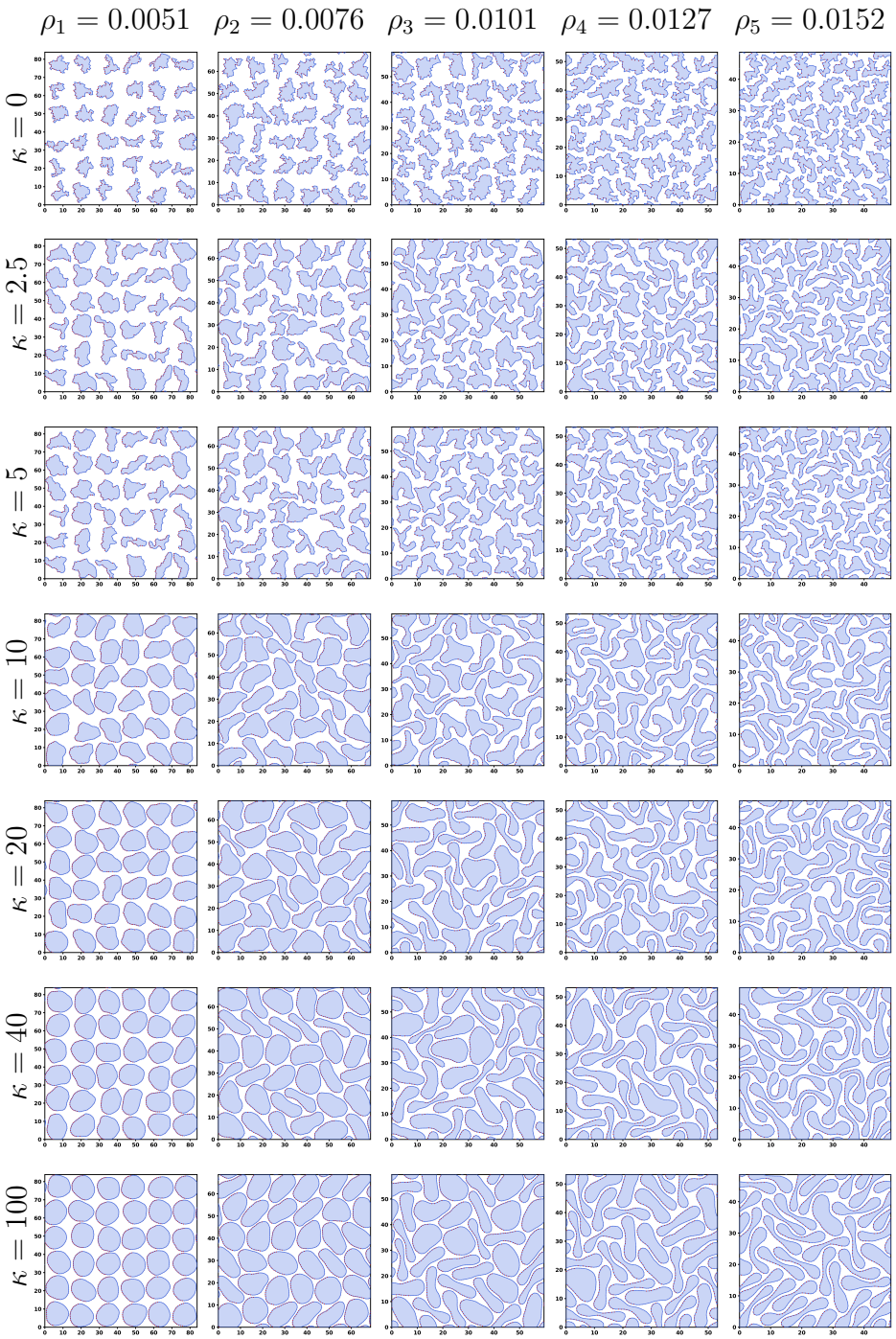


Figure 4.5: Matrix of steady-state cell conformations. Columns represent increasing areal density (ρ) from left to right. Rows represent increasing bending stiffness (κ) from top to bottom. A clear transition from rough, irregular shapes to smooth, biconcave structures is observed as κ increases and density rises.

Chapter 5

Result Analysis

In this chapter, we quantitatively analyze the conformational transitions observed in the simulations. We investigate how cell area and shape anisotropy evolve as a function of packing density (ρ) and bending rigidity (κ).

5.1 Conformational Behaviour

The qualitative transitions observed in Chapter 4 can be categorized into two distinct regimes driven by the competing energy scales of bending and repulsion.

1. **Effect of Bending Rigidity (κ):** At constant density, increasing κ smoothens the cell boundaries.
    - For low κ ($\kappa \approx 0$), cells exhibit "rough" edges dominated by thermal fluctuations.
    - For high κ ($\kappa \geq 40$), the high energy cost of curvature suppresses fluctuations, resulting in smooth, circular perimeters in the dilute limit.
2. **Effect of Density (ρ):** As the number density increases, steric repulsion between neighbors becomes the dominant force.
    - At low densities ($\rho < 0.0076 r_b^{-2}$), cells retain their equilibrium shapes (circular for high κ, fluctuating coils for low κ).
    - At high densities ($\rho > 0.0126 r_b^{-2}$), crowding forces a transition from circular to biconcave or anisotropic conformations to optimize packing efficiency.

Figure 5.1 summarizes these trends in a schematic phase diagram.

5.2 Mean Area Variation

To quantify the compression of cells under crowding, we track the mean encapsulated area $\langle a \rangle$ as a function of density. The area is calculated using the Shoelace formula:

$$A = \frac{1}{2} \left| \sum_{i=1}^{N} (x_i y_{i+1} - x_{i+1} y_i) \right| \tag{5.1}$$

The results (Fig. 5.2) indicate a monotonic decrease in mean area as density increases. Notably:

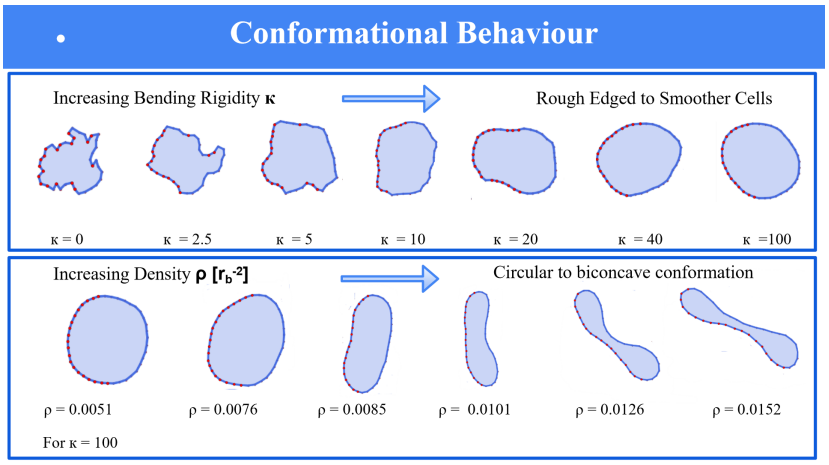


Figure 5.1: Conformational transitions as a function of bending rigidity κ and density ρ. The diagram illustrates the qualitative changes in cell shapes from rough coils to smooth circles and finally to elongated biconcave structures.

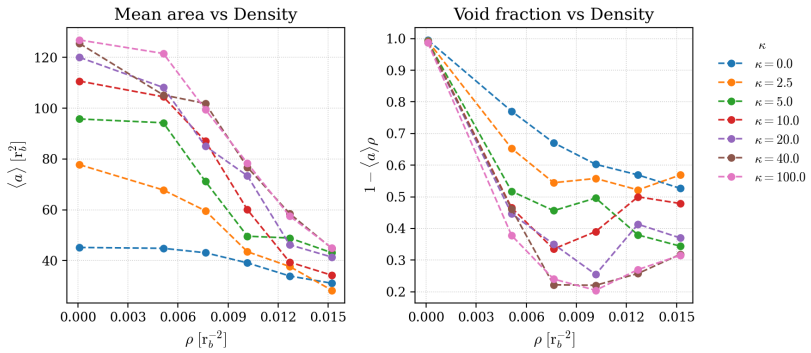


Figure 5.2: Variation of Mean Cell Area $\langle a \rangle$ with Density ρ for different bending stiffnesses κ.

- For flexible cells ($\kappa = 0$), the mean area is consistently lower due to entropic coiling.
- For rigid cells ($\kappa = 100$), the area remains close to the theoretical maximum for a circle at low densities but drops sharply as crowding forces the cells to deform and compress.

5.3 Order Parameter: Degree of Anisotropy

To distinguish between isotropic (circular) and anisotropic (elongated) shapes, we define an order parameter Σ based on the gyration tensor.

5.3.1 Definition

The gyration tensor $G_{\alpha\beta}$ for a ring polymer with N beads is defined as:

$$G_{\alpha\beta} = \frac{1}{N} \sum_{i=1}^{N} (\alpha_i - \alpha_{cm})(\beta_i - \beta_{cm}) \tag{5.2}$$

where $\alpha, \beta \in \{x, y\}$ are the Cartesian coordinates and $(\alpha_{cm}, \beta_{cm})$ is the center of mass. Diagonalizing $G_{\alpha\beta}$ yields two eigenvalues, λ_m (smaller) and λ_M (larger).

The degree of anisotropy Σ is defined as the ensemble average of the ratio of these eigenvalues:

$$\Sigma = \left\langle \frac{\lambda_m}{\lambda_M} \right\rangle \tag{5.3}$$

- $\Sigma \approx 1$ implies an isotropic, circular conformation.
- $\Sigma \to 0$ implies a highly anisotropic, rod-like conformation.

5.3.2 Quantitative Findings

Figure 5.3 plots Σ against density for various stiffness values.

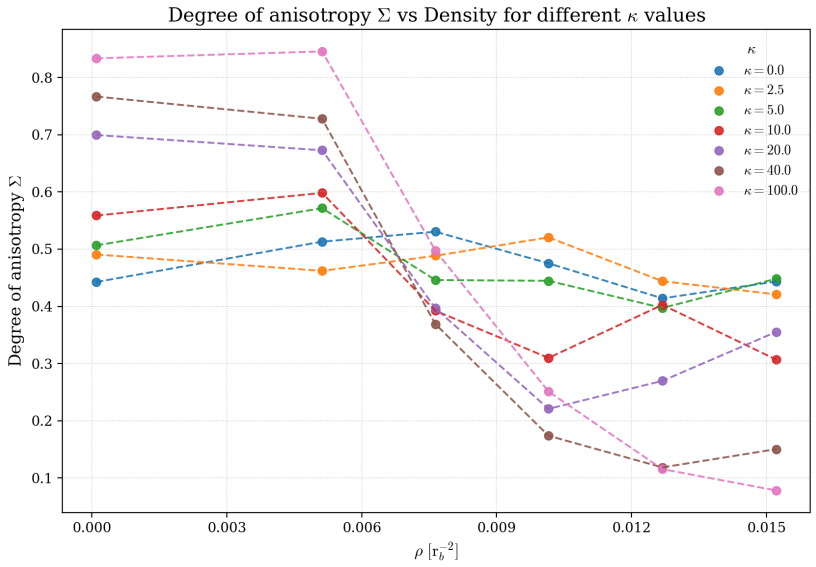


Figure 5.3: Degree of anisotropy Σ vs. Density ρ for different κ values.

The analysis reveals two contrasting behaviors:

1. **Low Stiffness ($\kappa < 5$):** The anisotropy is relatively stable or decreases slightly. These cells are already thermally deformed at low densities, so crowding has a less dramatic geometric effect.
2. **High Stiffness ($\kappa > 40$):** A sharp transition is observed. At low density, $\Sigma \approx 0.8$, indicating circular shapes. As density increases beyond $\rho \approx 0.009$, Σ plummets to < 0.4, quantitatively capturing the transition to elongated, biconcave structures driven by packing constraints.

Chapter 6

ML Algorithms and Architecture

In this chapter, we describe the machine learning frameworks developed to solve the inverse modeling problem. We explore two distinct architectures: a Convolutional Neural Network (CNN) that processes visual snapshots of the system, and a Graph Neural Network (GNN) that leverages the topological connectivity of the ring polymers.

6.1 Problem Statement

The core objective is to learn a mapping function f from the observable cell configurations (positions $\mathbf{r}$) to the underlying mechanical parameter, specifically the bending rigidity κ:

$$\kappa_{pred} = f(\mathbf{X}_{config}, \rho) \tag{6.1}$$

This inverse modeling approach enables the prediction of cell stiffness directly from static structural data, which is relevant for detecting invasive metastatic cells from microscopic images.

6.2 Convolutional Neural Network (CNN)

The first approach treats the simulation output as a visual image. The architecture fuses high-level visual features extracted from snapshots with explicit physical statistics (density, mean area).

6.2.1 Feature Extractor: MobileNetV2

We utilize **MobileNetV2**, a convolutional neural network pre-trained on the large-scale ImageNet dataset, as our primary feature extractor.

- **Transfer Learning:** although pre-trained on natural images, the fundamental features detected by the lower layers of MobileNetV2 (such as edges, curves, and textures) are highly transferable to the cellular boundaries in our simulations.
- **Frozen Weights:** The weights of the MobileNetV2 backbone are frozen to prevent overfitting on our smaller simulation dataset. This ensures that the model relies on robust, generalizable visual features.

6.2.2 Architecture Design and Feature Fusion

The complete pipeline consists of the following stages:

1. **Input Processing:** Grayscale simulation snapshots are converted to 3-channel RGB images to be compatible with the MobileNetV2 input requirements.

2. **Visual Feature Extraction:** The image passes through the frozen MobileNetV2 layers, followed by Global Average Pooling to flatten the spatial dimensions into a feature vector.

3. **Physical Feature Path:** In parallel, scalar physical statistics—specifically the *Mean Cell Area* and *System Density*—are processed through a small Multi-Layer Perceptron (MLP) with Batch Normalization.

4. **Feature Fusion:** The visual feature vector and the physical statistic vector are concatenated. This fusion allows the model to "reason" about the image context (e.g., crowding) using the explicit density values.

5. **Regression Head:** The combined vector passes through dense layers (with Dropout for regularization) to output a single linear prediction for κ.

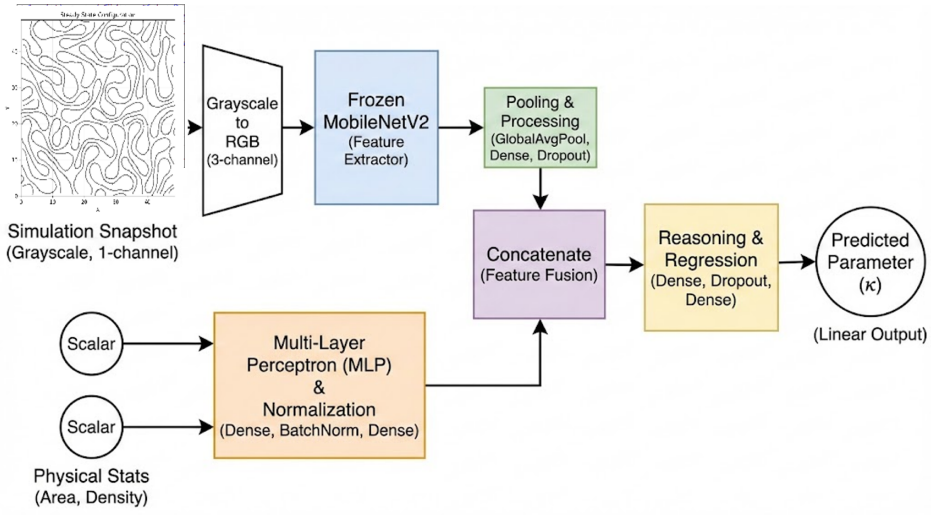


Figure 6.1: CNN Full Architecture showing the dual-pathway for Visual Features (MobileNetV2) and Physical Stats (MLP).

6.3 Graph Neural Network (GNN)

The second approach treats the cellular system as a graph, which is a more natural representation for polymers. This method captures the specific topology of the beads and springs.

6.3.1 Graph Construction

The simulation snapshot is converted into a graph data object $G = (V, E)$, where:

- **Nodes (V):** Each bead in the system represents a node.
- **Edges (E):** Edges are defined by the polymer connectivity. Bead i is connected to its neighbors $i-1$ and $i+1$ via the harmonic springs. This explicitly encodes the ring topology.

6.3.2 Node Feature Engineering

Instead of raw Cartesian coordinates (x, y), which are not rotationally invariant, we compute **Invariant Scalars** for every node i to serve as input features:

1. **Bond Length:** The distance to neighbors, $|r_{i,i\pm1}|$.
2. **Radial Distance:** The distance of the bead from the polymer's center of mass.
3. **Bond Angle:** The cosine of the angle θ_i formed by the triplet $(i-1, i, i+1)$.

These features ensure that the model prediction is invariant to the global rotation or translation of the system.

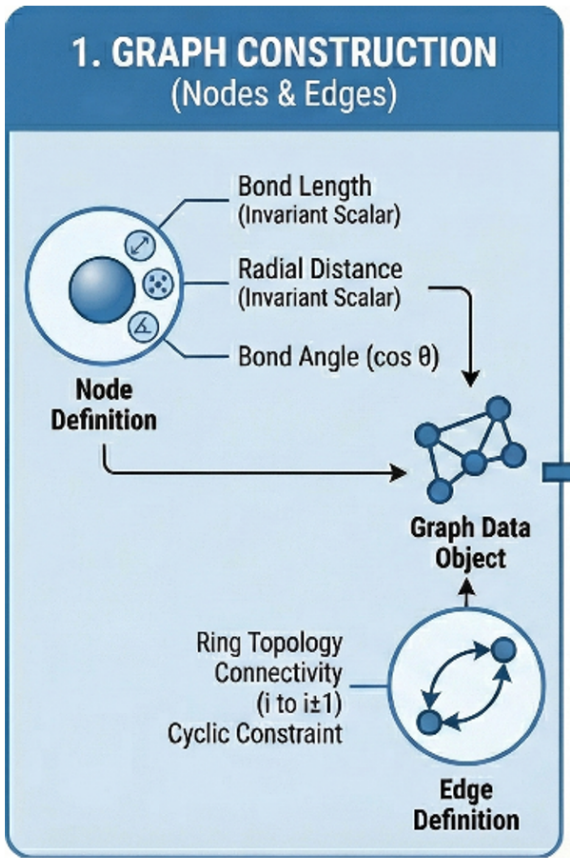


Figure 6.2: GNN Input Features: Bond Length, Radial Distance, and Bond Angle.

6.3.3 GNN Architecture

The GNN architecture processes the graph through message passing operations:

1. **GCN Layers:** The input features (dimension 3) are passed through three graph convolutional layers (GCNConv) with ReLU activation. The channel dimensions expand as $3 \to 64 \to 128 \to 128$, allowing the model to learn complex local geometric representations.

2. **Global Pooling (Readout):** To generate a single prediction for the entire system, node features are aggregated using a combination of *Global Mean*, *Global Max*, and *Global Std* pooling. This results in a comprehensive system-state vector of size 384.

3. **Regression Head:** The pooled vector is fed into a final MLP (Linear layers $384 \to 128 \to 64 \to 1$) to predict the scalar value κ.

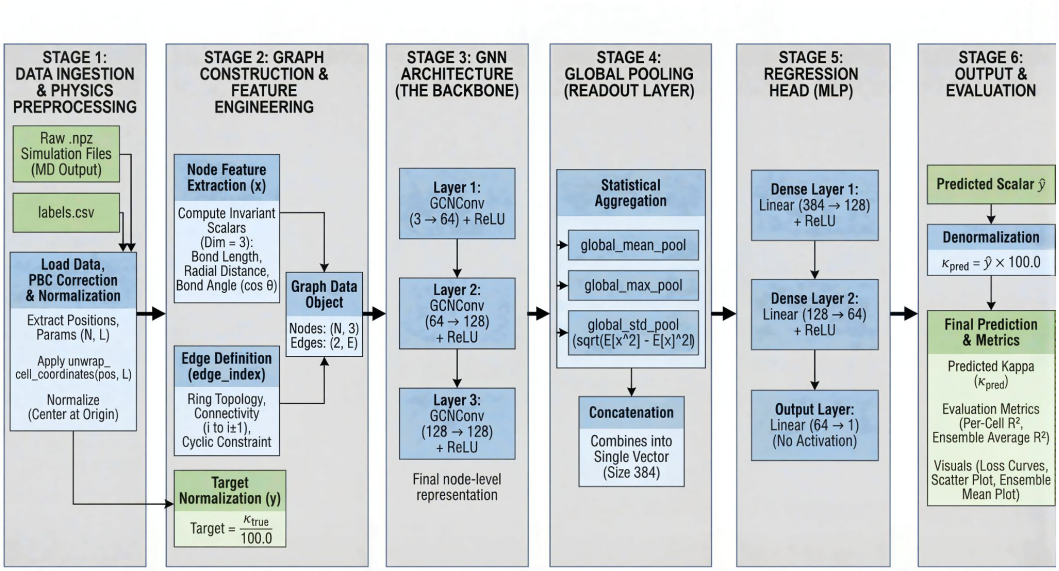


Figure 6.3: Full GNN Architecture showing Data Ingestion, Graph Construction, Backbone, and Readout layers.

Chapter 7

ML Results

In this chapter, we evaluate the performance of the trained machine learning models. We analyze the training stability via loss curves and quantify the predictive accuracy using the Coefficient of Determination (R^2) and Mean Absolute Error (MAE). We systematically compare the image-based CNN approach against the topology-based GNN approach through sample predictions and full dataset evaluations.

7.1 Convolutional Neural Network (CNN) Results

7.1.1 Training and Validation Performance

The CNN model was trained using the Huber loss function. Figure 7.1 shows the training trajectory over 20 epochs. While the loss decreases, the validation loss remains noisy. The corresponding prediction plot on the validation set (right) shows a reasonable correlation ($R^2 = 0.604$) but significant variance for higher stiffness values.

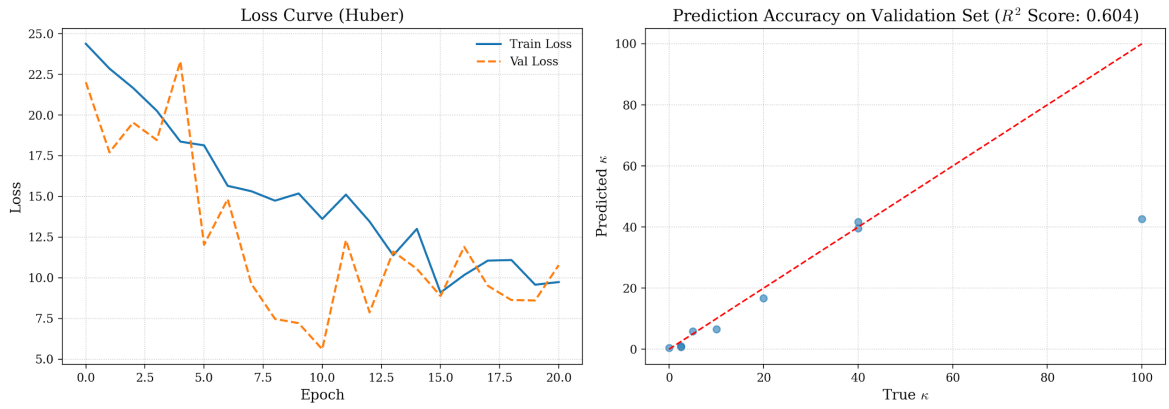


Figure 7.1: CNN Performance: (Left) Loss Curve showing training vs. validation loss. (Right) Prediction accuracy on the validation set, achieving an R^2 score of 0.604.

7.1.2 Sample Prediction Analysis

To understand the limitations of the CNN, we examine specific test samples. The model struggles significantly with density variations.

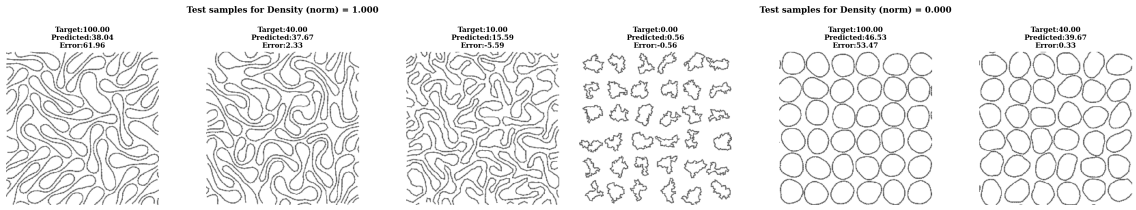


Figure 7.2: CNN Sample Predictions. (Left) At high density ($\rho \approx 1.0$), the model fails dramatically, predicting $\kappa \approx 38.04$ for a target of 100.00 (Error: 61.96). (Right) At low density ($\rho \approx 0.0$), prediction is also poor, estimating $\kappa \approx 46.53$ for a target of 100.00.

7.2 Graph Neural Network (GNN) Results

7.2.1 Training Stability and Validation

The GNN model demonstrates superior stability. The loss curve converges smoothly within 120 epochs. The validation performance is robust, with the single-cell prediction achieving an $R^2 = 0.887$ and the ensemble average reaching an impressive $R^2 = 0.964$.

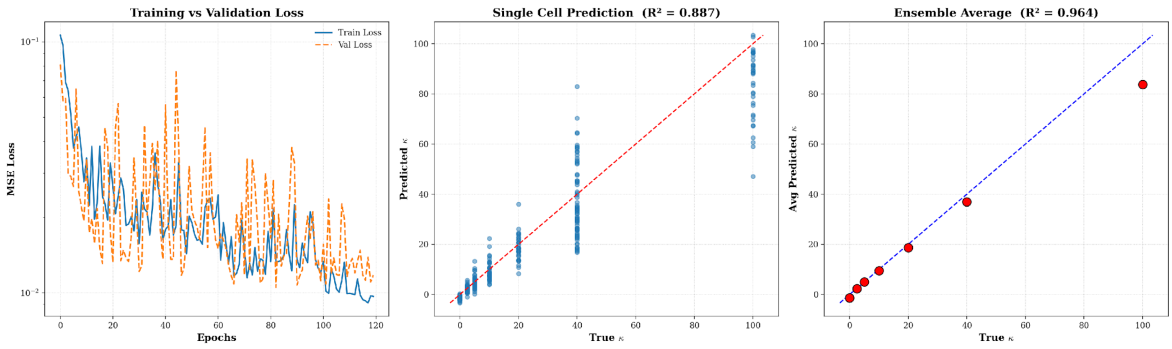


Figure 7.3: GNN Performance: (Left) Training vs. Validation Loss. (Center) Single-cell prediction accuracy ($R^2 = 0.887$). (Right) Ensemble average prediction accuracy ($R^2 = 0.964$).

7.2.2 Sample Prediction Analysis

The GNN generalizes much better across different density regimes compared to the CNN.(Figure 7.4).

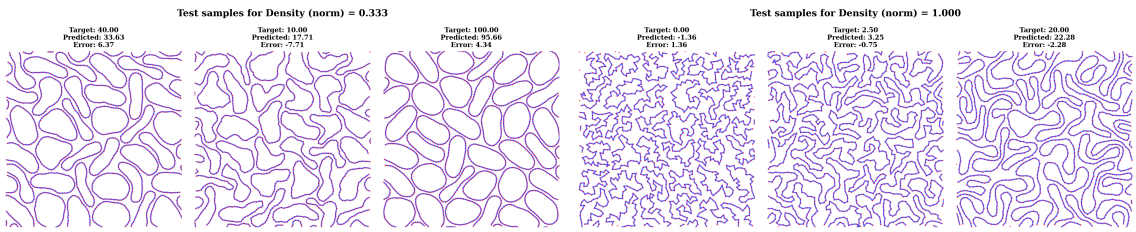


Figure 7.4: GNN Sample Predictions. (Left) At intermediate density ($\rho \approx 0.33$), the model predicts $\kappa \approx 95.66$ for a target of 100.00 (Error: 4.34). (Right) At high density ($\rho \approx 1.0$), the model accurately predicts $\kappa \approx 22.28$ for a target of 20.00 (Error: -2.28).

7.3 Full Dataset Evaluation and Comparison

Finally, we evaluate both models on the complete, separately generated dataset covering the full parameter space.

7.3.1 CNN Full Dataset Prediction

The CNN predictions are heavily stratified by density. As shown in Figure 7.5 (Left), different density groups (colored dots) form distinct clusters that often deviate from the ideal diagonal line. The overall performance is moderate ($R^2 \approx 0.544$) with high mean absolute error.

7.3.2 GNN Full Dataset Prediction

In contrast, the GNN predictions (Figure 7.5, Right) collapse onto the ideal diagonal regardless of the density. The ensemble averaging technique yields a high accuracy of $R^2 = 0.945$ with a low MAE of 4.067.

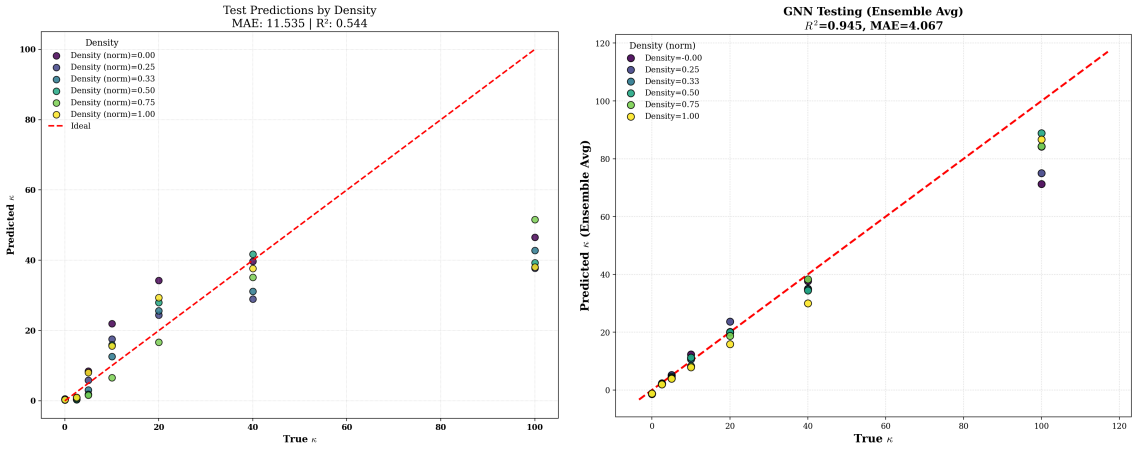


Figure 7.5: Overall Performance Comparison on Full Dataset. (Left) CNN predictions show significant density-dependent bias ($R^2 = 0.544$, MAE=11.535). (Right) GNN predictions are tightly clustered along the ideal line, demonstrating robustness across all densities ($R^2 = 0.945$, MAE=4.067).

Metric	CNN (Visual)	GNN (Topological)
Input Representation	2D Image (Pixels)	Graph (Nodes & Edges)
Validation R^2	0.604	**0.964** (Ensemble)
Full Dataset R^2	0.544	**0.945**
Full Dataset MAE	11.535	**4.067**

Table 7.1: Summary comparison of model performance metrics.

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

In this thesis, we have successfully integrated coarse-grained molecular dynamics (CGMD) simulations with advanced machine learning techniques to investigate the mechanosensitive behavior of collective cells. By modeling cells as semi-flexible ring polymers, we have established a quantitative link between single-cell mechanics and collective tissue morphology.

The key findings of this work can be summarized as follows:

8.1.1 Physics of Conformational Transitions

We demonstrated that the conformational behavior of ring polymers is governed by a delicate interplay between bending rigidity (κ) and packing density (ρ).

- **Conformational Transition:** We successfully captured the density-driven transition from circular equilibrium shapes to highly anisotropic, biconcave structures. This mimics the phenotypic plasticity observed in biological cells under confinement.
- **Rigidity & Density:** Our simulations confirmed that higher bending rigidity (κ) suppresses thermal fluctuations, leading to smoother cell boundaries. However, as density increases, even rigid cells are forced to deform, resulting in a monotonic decrease in the mean cell area.

8.1.2 Inverse Modeling via Machine Learning

We developed and compared two distinct machine learning architectures to solve the inverse problem of predicting cell stiffness from structural snapshots.

- **CNN Limitations:** The Convolutional Neural Network (CNN), while capable of fusing visual features with physical statistics, showed limited generalizability (κ prediction accuracy $R^2 \approx 0.604$). It struggled particularly with density-dependent variations.
- **High-Accuracy GNN:** The Graph Neural Network (GNN) proved superior by explicitly modeling the polymer topology. By treating beads as nodes and springs as edges, the GNN achieved a remarkable ensemble prediction accuracy

of $R^2 = 0.945$. This confirms that topological connectivity features are more robust descriptors of mechanical rigidity than raw visual textures.

8.2 Future Scope

While this work establishes a robust framework for mechano-inference, several avenues for future research remain to enhance the biological realism and practical applicability of the model.

8.2.1 Enhanced Potential Energy Functions

The current model relies on a minimal set of interactions (Spring, Bending, Repulsion). To better represent the complex potential landscape of biological tissues, future work should incorporate:

- **Area Conservation Potential:** Introducing an explicit area constraint potential to penalize deviations from a preferred cell volume, mimicking the incompressibility of the cytoplasm.
- **Lennard-Jones Potential:** Replacing the simple soft repulsion with a Lennard-Jones potential to model both short-range repulsion and long-range attraction (cell-cell adhesion).

8.2.2 Active Matter Dynamics

Biological cells are inherently non-equilibrium systems driven by internal energy consumption.

- **Motility Force:** We propose adding a self-propulsion force vector to each cell to model active migration. This would allow us to study collective motion phenomena such as jamming transitions and flocking.
- **Velocity Time-Series:** Utilizing velocity time-series data as an additional input for the ML models could help capture the dynamical signature of "active" particles, distinguishing them from passive Brownian motion.

8.2.3 Real-World Validation and Interpretability

- **Biological Validation:** The ultimate test of the inverse model is its application to actual biological systems. We aim to test the trained GNN on segmented microscopy images of epithelial tissues to predict cell stiffness values, validating them against AFM measurements.
- **Interpretability:** Moving away from the "black box" nature of deep learning, we aim to develop interpretable inference models that can explicitly relate geometric features (e.g., curvature distribution) to mechanical parameters.

Chapter 9

Codes

9.1 Simulation Code (Python/Numba)

```
from analysis_utility import *
from animation import *
from plot_state import *
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from numba import jit, njit, prange, float64, int32, types
from numba.types import List, Array
import numba

# Enable Numba parallel processing
os.environ['NUMBA_NUM_THREADS'] = str(os.cpu_count())

@njit(fastmath=True, cache=True)
def apply_pbc(positions, L):
    """Apply periodic boundary conditions to positions."""
    return positions % L

@njit(fastmath=True, cache=True)
def pbc_distance(r1, r2, L):
    """
    Calculate minimum distance between two points considering periodic boundaries.
    r1, r2: position vectors
    L: box size
    Returns: displacement vector from r1 to r2 with minimum image convention
    """
    dr = r2 - r1
    dr = dr - L * np.round(dr / L)
    return dr

@njit(fastmath=True, cache=True)
def velocity_verlet_step(pos, vel, force_func, Gamma, T, dt, L):
    b = 1.0 / (1 + Gamma * dt / 2)
    a = (1 - Gamma * dt / 2) * b
    sigma = np.sqrt(2 * Gamma * T * dt)

    # Numba-compatible random number generation
```

```
    noise = sigma * np.random.standard_normal(size=vel.shape)

    F = force_func(pos)
    pos_new = pos + b * vel * dt + (b * F / 2) * dt**2 + (b * noise / 2) * dt
    pos_new = apply_pbc(pos_new, L)

    F_new = force_func(pos_new)
    vel_new = a * vel + (dt / 2) * ((a * F + F_new)) + (b * noise)

    return pos_new, vel_new

@njit(fastmath=True, cache=True)
def ring_forces(positions, k=50.0, r0=1.0, L=20.0):
    """
    Compute spring forces between neighboring beads with periodic boundary conditions.
    """
    N = positions.shape[0]
    F = np.zeros_like(positions)
    for i in range(N):
        left_neighbor = (i - 1) % N
        right_neighbor = (i + 1) % N

        for j in (left_neighbor, right_neighbor):
            d = pbc_distance(positions[i], positions[j], L)
            r = np.sqrt(d[0]*d[0] + d[1]*d[1])
            if r > 1e-12:
                factor = k * (1.0 - r0 / r)
                F[i, 0] += factor * d[0]
                F[i, 1] += factor * d[1]
    return F

@njit(fastmath=True, cache=True)
def bending_forces(positions, kappa=1.0, L=20.0):
    """
    Compute bending forces on a ring of beads with periodic boundary conditions.
    """
    N = positions.shape[0]
    F = np.zeros_like(positions)

    for i in range(N):
        # neighbor indices (cyclic)
        im2 = (i - 2) % N
        im1 = (i - 1) % N
        ip1 = (i + 1) % N
        ip2 = (i + 2) % N

        # displacement vectors with PBC
        r_im1_i = pbc_distance(positions[i], positions[im1], L)
        r_ip1_i = pbc_distance(positions[i], positions[ip1], L)
        r_im2_im1 = pbc_distance(positions[im1], positions[im2], L)
        r_i_im1 = pbc_distance(positions[im1], positions[i], L)
        r_ip2_ip1 = pbc_distance(positions[ip1], positions[ip2], L)
        r_i_ip1 = pbc_distance(positions[ip1], positions[i], L)

        # norms squared
```

```
94          r_im1_i2 = r_im1_i[0]*r_im1_i[0] + r_im1_i[1]*r_im1_i[1]
95          r_ip1_i2 = r_ip1_i[0]*r_ip1_i[0] + r_ip1_i[1]*r_ip1_i[1]
96          r_im2_im1_2 = r_im2_im1[0]*r_im2_im1[0] + r_im2_im1[1]*
    r_im2_im1[1]
97          r_i_im1_2 = r_i_im1[0]*r_i_im1[0] + r_i_im1[1]*r_i_im1[1]
98          r_ip2_ip1_2 = r_ip2_ip1[0]*r_ip2_ip1[0] + r_ip2_ip1[1]*
    r_ip2_ip1[1]
99          r_i_ip1_2 = r_i_ip1[0]*r_i_ip1[0] + r_i_ip1[1]*r_i_ip1[1]
100
101         # dot products
102         dot_im1_ip1 = r_im1_i[0]*r_ip1_i[0] + r_im1_i[1]*r_ip1_i[1]
103         dot_im2_im1 = r_im2_im1[0]*r_i_im1[0] + r_im2_im1[1]*r_i_im1
    [1]
104         dot_ip2_ip1 = r_ip2_ip1[0]*r_i_ip1[0] + r_ip2_ip1[1]*r_i_ip1
    [1]
105
106         # Avoid division by zero
107         if (r_im1_i2 < 1e-12 or r_ip1_i2 < 1e-12 or r_im2_im1_2 < 1e
    -12 or
108             r_i_im1_2 < 1e-12 or r_ip2_ip1_2 < 1e-12 or r_i_ip1_2 < 1e
    -12):
109             continue
110
111         # Precompute square roots
112         sqrt_r_im1_i2 = np.sqrt(r_im1_i2)
113         sqrt_r_ip1_i2 = np.sqrt(r_ip1_i2)
114         sqrt_r_im2_im1_2 = np.sqrt(r_im2_im1_2)
115         sqrt_r_i_im1_2 = np.sqrt(r_i_im1_2)
116         sqrt_r_ip2_ip1_2 = np.sqrt(r_ip2_ip1_2)
117         sqrt_r_i_ip1_2 = np.sqrt(r_i_ip1_2)
118
119         # x component
120         if abs(dot_im1_ip1) > 1e-12:
121             term1_x = -(kappa * dot_im1_ip1 / (sqrt_r_im1_i2 *
    sqrt_r_ip1_i2)) * (
122                 -(r_im1_i[0] + r_ip1_i[0])/dot_im1_ip1 +
123                 r_im1_i[0]/r_im1_i2 +
124                 r_ip1_i[0]/r_ip1_i2
125             )
126         else:
127             term1_x = 0.0
128
129         if abs(dot_im2_im1) > 1e-12:
130             term2_x = -(kappa * dot_im2_im1 / (sqrt_r_im2_im1_2 *
    sqrt_r_i_im1_2)) * (
131                 r_im2_im1[0]/dot_im2_im1 -
132                 r_i_im1[0]/r_i_im1_2
133             )
134         else:
135             term2_x = 0.0
136
137         if abs(dot_ip2_ip1) > 1e-12:
138             term3_x = -(kappa * dot_ip2_ip1 / (sqrt_r_ip2_ip1_2 *
    sqrt_r_i_ip1_2)) * (
139                 r_ip2_ip1[0]/dot_ip2_ip1 -
140                 r_i_ip1[0]/r_i_ip1_2
141             )
142         else:
```

```
            term3_x = 0.0

        F[i, 0] = term1_x + term2_x + term3_x

        # y component
        if abs(dot_im1_ip1) > 1e-12:
            term1_y = -(kappa * dot_im1_ip1 / (sqrt_r_im1_i2 *
    sqrt_r_ip1_i2)) * (
                -(r_im1_i[1] + r_ip1_i[1])/dot_im1_ip1 +
                r_im1_i[1]/r_im1_i2 +
                r_ip1_i[1]/r_ip1_i2
            )
        else:
            term1_y = 0.0

        if abs(dot_im2_im1) > 1e-12:
            term2_y = -(kappa * dot_im2_im1 / (sqrt_r_im2_im1_2 *
    sqrt_r_i_im1_2)) * (
                r_im2_im1[1]/dot_im2_im1 -
                r_i_im1[1]/r_i_im1_2
            )
        else:
            term2_y = 0.0

        if abs(dot_ip2_ip1) > 1e-12:
            term3_y = -(kappa * dot_ip2_ip1 / (sqrt_r_ip2_ip1_2 *
    sqrt_r_i_ip1_2)) * (
                r_ip2_ip1[1]/dot_ip2_ip1 -
                r_i_ip1[1]/r_i_ip1_2
            )
        else:
            term3_y = 0.0

        F[i, 1] = term1_y + term2_y + term3_y

    return F



@njit(fastmath=True, cache=True, parallel=True)
def repulsion_forces_multi_cell(positions, cell_starts, cell_ends, chi
    =5.0, rm=1.0, L=20.0):
    """Parallel multi-cell repulsion focusing on inter-cell
    interactions"""
    n_cells = len(cell_starts)
    F = np.zeros_like(positions)
    rm_sq = rm * rm

    # Intra-cell repulsion (sequential - usually fast enough)
    for cell_idx in range(n_cells):
        start = cell_starts[cell_idx]
        end = cell_ends[cell_idx]
        N_cell = end - start

        for i_local in range(N_cell):
            i_global = start + i_local
            for j_local in range(i_local + 1, N_cell):
                j_global = start + j_local
```

```
                if j_local == (i_local - 1) % N_cell or j_local == (i_local + 1) % N_cell:
                    continue

                d = pbc_distance(positions[i_global], positions[j_global], L)
                r_sq = d[0]*d[0] + d[1]*d[1]

                if r_sq < 1e-24:
                    continue

                if r_sq <= rm_sq:
                    r = np.sqrt(r_sq)
                    factor = (chi / rm) * (1.0/rm - 1.0/r)
                    F[i_global, 0] += factor * d[0]
                    F[i_global, 1] += factor * d[1]
                    F[j_global, 0] -= factor * d[0]
                    F[j_global, 1] -= factor * d[1]

    # Inter-cell repulsion - PARALLEL over cell1_idx
    for cell1_idx in prange(n_cells):
        start1 = cell_starts[cell1_idx]
        end1 = cell_ends[cell1_idx]
        N_cell1 = end1 - start1

        for cell2_idx in range(cell1_idx + 1, n_cells):
            start2 = cell_starts[cell2_idx]
            end2 = cell_ends[cell2_idx]
            N_cell2 = end2 - start2

            for i_local in range(N_cell1):
                i_global = start1 + i_local
                for j_local in range(N_cell2):
                    j_global = start2 + j_local

                    d = pbc_distance(positions[i_global], positions[j_global], L)
                    r_sq = d[0]*d[0] + d[1]*d[1]

                    if r_sq < 1e-24:
                        continue

                    if r_sq <= rm_sq:
                        r = np.sqrt(r_sq)
                        factor = (chi / rm) * (1.0/rm - 1.0/r)
                        # Use atomic operations to avoid race conditions
                        # Numba handles this automatically with prange
                        F[i_global, 0] += factor * d[0]
                        F[i_global, 1] += factor * d[1]
                        F[j_global, 0] -= factor * d[0]
                        F[j_global, 1] -= factor * d[1]

    return F

@njit(fastmath=True, cache=True)
def polygon_area_pbc(positions, L):
    """Compute signed areagon of poly with periodic boundary
```

```
    conditions"""
    N = positions.shape[0]
    area = 0.0

    for i in range(N):
        j = (i + 1) % N
        dr = pbc_distance(positions[i], positions[j], L)
        next_pos = positions[i] + dr
        area += positions[i, 0] * next_pos[1] - next_pos[0] *
    positions[i, 1]

    return 0.5 * abs(area)

@njit(fastmath=True, cache=True)
def area_forces(positions, A0, phi=1.0, L=20.0):
    """Compute area forces on a ring of beads with periodic boundary
    conditions"""
    N = positions.shape[0]
    F = np.zeros_like(positions)

    # compute current area with PBC
    A = polygon_area_pbc(positions, L)

    # prefactor
    pref = 0.5 * phi * (1 - A/A0)

    for i in range(N):
        im1 = (i - 1) % N
        ip1 = (i + 1) % N

        # Use PBC distance to get proper displacement vectors
        dr_im1 = pbc_distance(positions[i], positions[im1], L)
        dr_ip1 = pbc_distance(positions[i], positions[ip1], L)

        # Reconstruct neighbor positions relative to current position
        pos_im1_rel = positions[i] + dr_im1
        pos_ip1_rel = positions[i] + dr_ip1

        F[i, 0] = pref * (pos_im1_rel[1] - pos_ip1_rel[1])
        F[i, 1] = pref * (pos_ip1_rel[0] - pos_im1_rel[0])

    return F



@njit(fastmath=True, cache=True)
def total_forces_multi_cell(positions, cell_starts, cell_ends, L=20.0,
    k=50.0, r0=1.0, kappa=1.0,
                            chi=5.0, rm=1.0, A0=None, phi=1.0):
    """Compiled total forces for multi-cell case with Numba-compatible
    indexing"""
    n_cells = len(cell_starts)
    F_total = np.zeros_like(positions)

    # Process each cell's internal forces
    for cell_idx in range(n_cells):
        start = cell_starts[cell_idx]
        end = cell_ends[cell_idx]
```

```
303         cell_pos = positions[start:end]
304         F_cell = np.zeros_like(cell_pos)
305 
306         F_cell += ring_forces(cell_pos, k=k, r0=r0, L=L)
307         F_cell += bending_forces(cell_pos, kappa=kappa, L=L)
308 
309         # Handle area forces with A0
310         if A0 is not None:
311             # For multi-cell, use first A0 value for all cells if not
    a list
312             A0_cell = A0
313             F_cell += area_forces(cell_pos, A0=A0_cell, phi=phi, L=L)
314 
315         F_total[start:end] += F_cell
316 
317     # Add repulsion forces between all cells
318     F_total += repulsion_forces_multi_cell(positions, cell_starts,
    cell_ends, chi=chi, rm=rm, L=L)
319 
320     return F_total
321 
322 @njit(fastmath=True, cache=True)
323 def get_grid_centers(n_cells, L):
324     """Get centers for placing cells in a grid - Numba compatible"""
325     grid_size = int(np.ceil(np.sqrt(n_cells)))
326     spacing = L / grid_size
327 
328     centers = np.zeros((grid_size * grid_size, 2))
329     idx = 0
330     for i in range(grid_size):
331         for j in range(grid_size):
332             centers[idx, 0] = (i + 0.5) * spacing
333             centers[idx, 1] = (j + 0.5) * spacing
334             idx += 1
335 
336     return centers, grid_size
337 
338 def simulate_cell(
339     N=20, L=20.0, Rinit=1.0, k=50.0, r0=1.0,
340     Gamma=1.0, T=0.0, dt=1e-3, steps=5000, seed=0,
341     int_noise=0.0, kappa=1.0, chi=5.0, rm=1.0,
342     A0=None, phi=1.0, save=True, n_cells=1, center_noise=0
343 ):
344     """
345     Optimized multi-cell simulation with Numba acceleration
346     """
347     params = {
348         "N": N, "L": L, "Rinit": Rinit, "k": k, "r0": r0,
349         "Gamma": Gamma, "T": T, "dt": dt,
350         "steps": steps, "seed": seed, "i_n": int_noise, "kappa": kappa
    ,
351         "chi": chi, "rm": rm, "A0": A0, "phi": phi, "n_cells": n_cells
    ,
352         "center_noise": center_noise
353     }
354 
355 
356     # Initialize random state for Numba compatibility
```

```
    np.random.seed(seed)

    # Get grid centers for cell placement
    grid_centers, grid_size = get_grid_centers(n_cells, L)

    # If more centers than needed, randomly select
    if len(grid_centers) > n_cells:
        selected_indices = np.random.choice(len(grid_centers), size=n_cells, replace=False)
        grid_centers = grid_centers[selected_indices]

    # Add noise to centers
    if center_noise > 0:
        grid_centers += np.random.normal(0.0, center_noise, size=grid_centers.shape)
        grid_centers = apply_pbc(grid_centers, L)

    # Initialize all positions and create Numba-compatible cell indexing
    N_total = N * n_cells
    pos = np.zeros((N_total, 2))

    # Use start/end indices instead of list of arrays for Numba compatibility
    cell_starts = np.zeros(n_cells, dtype=np.int32)
    cell_ends = np.zeros(n_cells, dtype=np.int32)




    for cell_idx in range(n_cells):
        start_idx = cell_idx * N
        end_idx = (cell_idx + 1) * N
        cell_starts[cell_idx] = start_idx
        cell_ends[cell_idx] = end_idx

        # Create circular arrangement for this cell
        theta = np.linspace(0, 2*np.pi, N, endpoint=False)
        cell_center = grid_centers[cell_idx]

        cell_pos = cell_center + np.column_stack((np.cos(theta), np.sin(theta))) * Rinit

        # Add individual bead noise
        if int_noise > 0:
            cell_pos += np.random.normal(0.0, int_noise, size=cell_pos.shape)

        cell_pos = apply_pbc(cell_pos, L)
        pos[start_idx:end_idx] = cell_pos

    vel = np.zeros((N_total, 2))

    positions = np.zeros((steps, N_total, 2))
    velocities = np.zeros((steps, N_total, 2))
    positions[0] = pos
    velocities[0] = vel

```

```
409
410     # Use the multi-cell version with Numba-compatible indexing
411     force_func_numba = njit(fastmath=True, cache=True)(
412         lambda p: total_forces_multi_cell(p, cell_starts, cell_ends, L
    =L, k=k, r0=r0,
413                                           kappa=kappa, chi=chi, rm=rm,
    A0=A0, phi=phi)
414     )
415
416     # Warm-up compilation
417     _ = force_func_numba(pos)
418
419     # Main simulation loop
420     for s in range(1, steps):
421         pos, vel = velocity_verlet_step(
422             pos, vel, force_func_numba, Gamma, T, dt, L
423         )
424
425         positions[s] = pos
426         velocities[s] = vel
427
428     # Reconstruct cell_indices for output
429     cell_indices = [np.arange(cell_starts[i], cell_ends[i]) for i in
    range(n_cells)]
430
431     if save:
432         keys_to_include = ["kappa", "chi", "rm", "phi", "n_cells","
    steps","L","A0"]
433         filename_base = "_".join(f"{key}-{params[key]}" for key in
    keys_to_include)
434         filename_base = filename_base.replace(".", "p").replace(" ", "
    ")[:100]
435
436         os.makedirs("simulation_output", exist_ok=True)
437         positions_save = positions.astype(np.float32)
438         velocities_save = velocities.astype(np.float32)
439
440         np.savez_compressed(f"simulation_output/{filename_base}.npz",
441                             positions=positions_save,
442                             velocities=velocities_save,
443                             params=params,
444                             cell_indices=cell_indices)
445     return positions, velocities, params, cell_indices
446
447 # Pre-compilation function
448 def precompile_functions():
449     """Pre-compile all Numba functions for better performance"""
450     print("Pre-compiling Numba functions...")
451
452     # Test data
453     test_pos = np.random.random((20, 2))
454     test_L = 20.0
455     test_cell_starts = np.array([0, 10], dtype=np.int32)
456     test_cell_ends = np.array([10, 20], dtype=np.int32)
457
458     # Compile all functions
459     _ = apply_pbc(test_pos, test_L)
460     _ = pbc_distance(test_pos[0], test_pos[1], test_L)
```

```
    _ = ring_forces(test_pos, L=test_L)
    _ = bending_forces(test_pos, L=test_L)
    _ = repulsion_forces_multi_cell(test_pos, test_cell_starts,
    test_cell_ends, L=test_L)
    _ = polygon_area_pbc(test_pos, test_L)
    _ = area_forces(test_pos, A0=10.0, L=test_L)
    _ = total_forces_multi_cell(test_pos, test_cell_starts,
    test_cell_ends, L=test_L)
    _ = get_grid_centers(4, test_L)

    print("Pre-compilation complete!")

# Pre-compile on module import
precompile_functions()
```

Listing 9.1: MD Simulation Code

9.2 CNN Model Code (TensorFlow)

```
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Dense, Dropout,
    GlobalAveragePooling2D, Input, Concatenate, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping,
    ReduceLROnPlateau
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt

# =========================================================
# 1. Configuration
# =========================================================
IMG_SIZE = (224, 224)
BATCH_SIZE = 8
EPOCHS = 50
DATA_DIR = "Final_Data"
SCALAR_FEATURES = ['mean_area', 'density']
RANDOM_SEED = 42 # Fixed Seed for reproducibility

# =========================================================
# 2. Load Data & Create Stratified Split
# =========================================================
csv_path = os.path.join(DATA_DIR, "labels.csv")

if not os.path.exists(csv_path):
    raise FileNotFoundError(f"Error: Could not find {csv_path}")

df = pd.read_csv(csv_path)

# Convert filename to full path for loading
df['full_path'] = df['filename'].astype(str).apply(lambda x: os.path.
    join(DATA_DIR, "images", x))

```

```
print(f"Total samples found: {len(df)}")

# --- STRATIFICATION LOGIC ---
# Since kappa is continuous, we create temporary bins to stratify against
# This ensures both Train and Val have a mix of high, medium, and low kappa values
df['kappa_bin'] = pd.qcut(df['kappa'], q=10, labels=False, duplicates='drop')

train_df, val_df = train_test_split(
    df,
    test_size=0.2,
    random_state=RANDOM_SEED,
    stratify=df['kappa_bin'] # Stratify based on the bins
)

# Clean up temporary bin column
train_df = train_df.drop(columns=['kappa_bin'])
val_df = val_df.drop(columns=['kappa_bin'])

# --- CRITICAL: SAVE VALIDATION SET FOR GNN ---
# We save the filenames of the validation set so the GNN can load the EXACT same files.
val_filename_save_path = "common_validation_split.csv"
val_df[['filename']].to_csv(val_filename_save_path, index=False)
print(f"Validation split saved to {val_filename_save_path}. Use this file for the GNN.")

print(f"Training Samples: {len(train_df)}")
print(f"Validation Samples: {len(val_df)}")

# ==========================================================
# 3. Data Normalization (Fit on TRAIN, Apply to ALL)
# ==========================================================
# Prevent data leakage by only calculating min/max on training set
stats = {}
for feature in SCALAR_FEATURES:
    min_val = train_df[feature].min()
    max_val = train_df[feature].max()
    denom = max_val - min_val + 1e-7

    # Store stats to reuse if you deploy the model later
    stats[feature] = {'min': min_val, 'denom': denom}

    # Apply normalization
    train_df[feature] = (train_df[feature] - min_val) / denom
    val_df[feature] = (val_df[feature] - min_val) / denom

# ==========================================================
# 4. Generators
# ==========================================================
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Augmentation only for training
train_datagen_config = ImageDataGenerator(
    rescale=1./255,
    rotation_range=180,
```

```
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode='nearest'
)
val_datagen_config = ImageDataGenerator(rescale=1./255)

class PhysicsDataGenerator:
    def __init__(self, dataframe, image_datagen, batch_size, img_size):
        self.dataframe = dataframe.reset_index(drop=True)
        self.batch_size = batch_size
        self.img_size = img_size
        self.image_datagen = image_datagen
        self.n = len(dataframe)
        self.indices = np.arange(self.n)
        self.on_epoch_end()

    def on_epoch_end(self):
        np.random.shuffle(self.indices)

    def __len__(self):
        return int(np.ceil(self.n / self.batch_size))

    def __call__(self):
        current_index = 0
        while True:
            if current_index >= self.n:
                current_index = 0
                self.on_epoch_end()

            batch_indices = self.indices[current_index:current_index + self.batch_size]
            batch_df = self.dataframe.iloc[batch_indices]

            # Process Images
            batch_images = []
            for full_path in batch_df['full_path']:
                img = tf.keras.preprocessing.image.load_img(full_path, color_mode='grayscale',
    target_size=self.img_size)
                img_array = tf.keras.preprocessing.image.img_to_array(img)
                batch_images.append(img_array)

            batch_images = np.array(batch_images)
            batch_images = self.image_datagen.standardize(batch_images)

            # Process Scalars
            batch_scalars = batch_df[SCALAR_FEATURES].values

            # Process Targets
            batch_targets = batch_df['kappa'].values.reshape(-1, 1)

            current_index += self.batch_size
            yield ({'image_input': batch_images, 'scalar_input':
```

```
    batch_scalars}, batch_targets)

train_generator = PhysicsDataGenerator(train_df, train_datagen_config,
    BATCH_SIZE, IMG_SIZE)
val_generator = PhysicsDataGenerator(val_df, val_datagen_config,
    BATCH_SIZE, IMG_SIZE)

# =======================================================
# 5. Model Architecture
# =======================================================
def create_physics_model(img_size):
    # --- Image Branch ---
    image_input = Input(shape=(img_size[0], img_size[1], 1), name='
    image_input')
    x = Concatenate(axis=-1)([image_input] * 3) # MobileNet needs 3
    channels

    base_model = MobileNetV2(input_shape=(img_size[0], img_size[1], 3)
    , include_top=False, weights="imagenet")
    base_model.trainable = False

    x = base_model(x)
    x = GlobalAveragePooling2D()(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.4)(x)

    # --- Scalar Branch ---
    scalar_input = Input(shape=(len(SCALAR_FEATURES),), name='
    scalar_input')
    s = Dense(32, activation='relu')(scalar_input)
    s = BatchNormalization()(s)
    s = Dense(32, activation='relu')(s)

    # --- Fusion ---
    combined = Concatenate()([x, s])

    z = Dense(32, activation='relu')(combined)
    z = Dropout(0.3)(z)
    z = Dense(32, activation='relu')(z)

    output = Dense(1, activation='linear', name='kappa_output')(z)

    model = Model(inputs=[image_input, scalar_input], outputs=output)
    return model

model = create_physics_model(IMG_SIZE)
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    loss=tf.keras.losses.Huber(),
    #loss = 'mse',
    metrics=['mae', 'mse']
)

# =======================================================
# 6. Training
# =======================================================
steps_per_epoch = int(np.ceil(len(train_df) / BATCH_SIZE))
validation_steps = int(np.ceil(len(val_df) / BATCH_SIZE))
```

```
191
192 early_stop = EarlyStopping(monitor='val_loss', patience=10,
    restore_best_weights=True)
193 reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience
    =5, min_lr=1e-6)
194
195 history = model.fit(
196     train_generator(),
197     validation_data=val_generator(),
198     epochs=EPOCHS,
199     steps_per_epoch=steps_per_epoch,
200     validation_steps=validation_steps,
201     callbacks=[early_stop, reduce_lr],
202     verbose=1
203 )
204
205 # ========================================================
206 # 7. Results
207 # ========================================================
208 def get_predictions(gen_func, steps):
209     instance = gen_func()
210     trues, preds = [], []
211     for _ in range(steps):
212         batch = next(instance)
213         p = model.predict(batch[0], verbose=0)
214         trues.extend(batch[1].flatten())
215         preds.extend(p.flatten())
216     return np.array(trues), np.array(preds)
217
218 y_true, y_pred = get_predictions(val_generator, validation_steps)
219
220 # R2 Score Calculation
221 from sklearn.metrics import r2_score
222 r2 = r2_score(y_true, y_pred)
223 print(f"Final Validation R2 Score: {r2:.4f}")
224
225 plt.figure(figsize=(14, 5))
226 plt.subplot(1, 2, 1)
227 plt.plot(history.history['loss'], label='Train')
228 plt.plot(history.history['val_loss'], label='Val')
229 plt.title('Loss')
230 plt.legend()
231
232 plt.subplot(1, 2, 2)
233 plt.scatter(y_true, y_pred, alpha=0.6)
234 plt.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], '
    r--')
235 plt.title(f'True vs Pred (R2: {r2:.3f})')
236 plt.show()
```

Listing 9.2: CNN Architecture Code

9.3 GNN Model Code (PyTorch)

```
1 import numpy as np
2 import pandas as pd
```

```
import torch
import torch.nn as nn
from torch_geometric.data import Data, DataLoader
from torch_geometric.nn import GCNConv, global_mean_pool,
    global_max_pool
from sklearn.metrics import r2_score, mean_absolute_error
import os
import matplotlib.pyplot as plt
from cycler import cycler

# ==========================================
# 1. CONFIGURATION & SEED
# ==========================================
SEED = 42
torch.manual_seed(SEED)
np.random.seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)

# UPDATE THESE PATHS IF NECESSARY
SIM_OUTPUT_DIR = r"D:\Thesis3\simulation_output"
CSV_PATH = r"Final_Data\labels.csv"
VALIDATION_SPLIT_PATH = "common_validation_split.csv"



# ==========================================
# 2. PHYSICS HELPER FUNCTIONS
# ==========================================
def unwrap_cell_coordinates(cell_pos, L):
    diff = cell_pos[1:] - cell_pos[:-1]
    diff = diff - L * np.round(diff / L)
    unwrapped = np.zeros_like(cell_pos)
    unwrapped[0] = cell_pos[0]
    for i in range(len(diff)):
        unwrapped[i+1] = unwrapped[i] + diff[i]
    return unwrapped - np.mean(unwrapped, axis=0)

def compute_invariant_features(pos):
    N = len(pos)
    features = []
    for i in range(N):
        prev_idx = (i - 1) % N
        curr_idx = i
        next_idx = (i + 1) % N

        u = pos[curr_idx] - pos[prev_idx]
        v = pos[next_idx] - pos[curr_idx]

        bond_len = np.linalg.norm(u)
        rad_dist = np.linalg.norm(pos[curr_idx])

        norm_u = np.linalg.norm(u)
        norm_v = np.linalg.norm(v)
        if norm_u == 0 or norm_v == 0:
            cos_theta = 0.0
        else:
            cos_theta = np.dot(u, v) / (norm_u * norm_v)
```

```
60
61          features.append([bond_len, rad_dist, cos_theta])
62      return np.array(features)
63
64  # ==========================================
65  # 3. DATASET CLASS
66  # ==========================================
67  class RingPolymerDataset(torch.utils.data.Dataset):
68      def __init__(self, target_filenames, labels_map, root_dir):
69          self.data_list = []
70          self.MAX_KAPPA = 100.0
71
72          print(f"Loading {len(target_filenames)} files into dataset..."
    )
73
74          for fname in target_filenames:
75              file_path = os.path.join(root_dir, fname)
76
77              if not os.path.exists(file_path):
78                  continue
79              if fname not in labels_map:
80                  continue
81
82              try:
83                  with np.load(file_path, allow_pickle=True) as data:
84                      params = data['params'].item()
85                      positions_steady = data['positions'][-1]
86
87                  kappa = labels_map[fname]
88                  y_norm = kappa / self.MAX_KAPPA
89
90                  N_beads = params['N']
91                  N_cells = params['n_cells']
92                  L = params['L']
93
94                  for c in range(N_cells):
95                      start = c * N_beads
96                      end = (c + 1) * N_beads
97
98                      raw_pos = positions_steady[start:end]
99                      clean_pos = unwrap_cell_coordinates(raw_pos, L)
100
101                     feats = compute_invariant_features(clean_pos)
102                     x = torch.tensor(feats, dtype=torch.float)
103
104                     edges = []
105                     for i in range(N_beads):
106                         edges.append([i, (i+1)%N_beads])
107                         edges.append([(i+1)%N_beads, i])
108                     edge_index = torch.tensor(edges, dtype=torch.long)
    .t().contiguous()
109
110                     y = torch.tensor([y_norm], dtype=torch.float)
111                     self.data_list.append(Data(x=x, edge_index=
    edge_index, y=y))
112
113             except Exception as e:
114                 print(f"Error processing {fname}: {e}")
```

```
115
116     def __len__(self):
117         return len(self.data_list)
118
119     def __getitem__(self, idx):
120         return self.data_list[idx]
121
122 # ==========================================
123 # 4. GNN MODEL
124 # ==========================================
125 class RingGNN(torch.nn.Module):
126     def __init__(self):
127         super(RingGNN, self).__init__()
128         self.conv1 = GCNConv(3, 64)
129         self.conv2 = GCNConv(64, 128)
130         self.conv3 = GCNConv(128, 128)
131
132         self.lin1 = torch.nn.Linear(128 * 3, 128)
133         self.lin2 = torch.nn.Linear(128, 64)
134         self.lin3 = torch.nn.Linear(64, 1)
135
136     def forward(self, x, edge_index, batch):
137         x = self.conv1(x, edge_index).relu()
138         x = self.conv2(x, edge_index).relu()
139         x = self.conv3(x, edge_index).relu()
140
141         x_mean = global_mean_pool(x, batch)
142         x_max = global_max_pool(x, batch)
143         x_sq_mean = global_mean_pool(x**2, batch)
144         x_var = (x_sq_mean - x_mean**2).clamp(min=1e-6)
145         x_std = torch.sqrt(x_var)
146
147         x_cat = torch.cat([x_mean, x_max, x_std], dim=1)
148
149         x = self.lin1(x_cat).relu()
150         x = self.lin2(x).relu()
151         x = self.lin3(x)
152         return x
153
154 # ==========================================
155 # 5. EXECUTION PIPELINE
156 # ==========================================
157 def run_pipeline():
158     # 1. Load Master Labels
159     if not os.path.exists(CSV_PATH):
160         print(f"Error: {CSV_PATH} not found.")
161         return None
162
163     labels_df = pd.read_csv(CSV_PATH)
164     labels_df['filename'] = labels_df['filename'].astype(str).str.
    replace('.png', '.npz', regex=False)
165     labels_df['filename'] = labels_df['filename'].str.replace('
    _steady_state', '', regex=False)
166
167     label_map = dict(zip(labels_df['filename'], labels_df['kappa']))
168
169     # 2. Load Validation Split from CNN
170     if not os.path.exists(VALIDATION_SPLIT_PATH):
```

```
        print(f"Error: {VALIDATION_SPLIT_PATH} not found. Run CNN code first!")
        return None

    val_split_df = pd.read_csv(VALIDATION_SPLIT_PATH)
    val_files_raw = val_split_df['filename'].astype(str).tolist()
    val_files_npz = [f.replace('.png', '.npz').replace('_steady_state', '') for f in val_files_raw]

    # 3. Create Train/Val Lists
    all_files = labels_df['filename'].tolist()
    val_set = set(val_files_npz)

    train_files_npz = [f for f in all_files if f not in val_set]

    print(f"Total Files: {len(all_files)}")
    print(f"Training Files: {len(train_files_npz)}")
    print(f"Validation Files: {len(val_files_npz)}")

    # 4. Create Datasets
    train_dataset = RingPolymerDataset(train_files_npz, label_map, SIM_OUTPUT_DIR)
    test_dataset = RingPolymerDataset(val_files_npz, label_map, SIM_OUTPUT_DIR)

    # 5. Loaders
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Training on: {device}")

    model = RingGNN().to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    criterion = torch.nn.MSELoss()

    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', factor=0.5, patience=15
    )

    print("Starting Training...")
    best_val_loss = float('inf')
    best_model_path = "best_ring_gnn.pth"
    history = {'train_loss': [], 'val_loss': []}

    epochs = 120

    for epoch in range(1, epochs + 1):
        model.train()
        total_train_loss = 0
        for data in train_loader:
            data = data.to(device)
            optimizer.zero_grad()
            out = model(data.x, data.edge_index, data.batch)
            loss = criterion(out.squeeze(), data.y)
```

```
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item() * data.num_graphs

        avg_train_loss = total_train_loss / len(train_dataset)

        model.eval()
        total_val_loss = 0
        with torch.no_grad():
            for data in test_loader:
                data = data.to(device)
                out = model(data.x, data.edge_index, data.batch)
                loss = criterion(out.squeeze(), data.y)
                total_val_loss += loss.item() * data.num_graphs

        avg_val_loss = total_val_loss / len(test_dataset)

        history['train_loss'].append(avg_train_loss)
        history['val_loss'].append(avg_val_loss)

        scheduler.step(avg_val_loss)

        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            torch.save(model.state_dict(), best_model_path)

        if epoch % 10 == 0:
            lr = optimizer.param_groups[0]['lr']
            print(f"Epoch {epoch:03d} | Train MSE: {avg_train_loss:.5f
    } | Val MSE: {avg_val_loss:.5f} | LR: {lr:.6f}")

    # 6. Evaluation
    print(f"\nLoading best model (Loss: {best_val_loss:.5f})...")
    model.load_state_dict(torch.load(best_model_path))
    model.eval()

    preds = []
    actuals = []

    with torch.no_grad():
        for data in test_loader:
            data = data.to(device)
            out = model(data.x, data.edge_index, data.batch)
            preds.extend(out.squeeze().cpu().numpy() * 100.0)
            actuals.extend(data.y.cpu().numpy() * 100.0)

    df_res = pd.DataFrame({'True': actuals, 'Pred': preds})

    # --- Metrics Calculation for Return Dict ---
    r2_cell = r2_score(actuals, preds)
    mae_avg = mean_absolute_error(actuals, preds)

    # Calculate Ensemble Metrics (Averaged by True value)
    df_avg = df_res.groupby('True').mean().reset_index()
    r2_avg = r2_score(df_avg['True'], df_avg['Pred'])

    print(f"\n=== FINAL RESULTS (VALIDATION SET) ===")
    print(f"Per Cell R2: {r2_cell:.4f}")
```

```
    print(f"Ensemble Avg R2: {r2_avg:.4f} (MAE: {mae_avg:.4f})")

    # --- FINAL RETURN DICTIONARY ---
    return {
        'history': history,
        'df_res': df_res,
        'r2_cell': r2_cell,
        'r2_avg': r2_avg,
        'mae_avg': mae_avg,
        'best_val_loss': best_val_loss
    }

if __name__ == "__main__":
    results = run_pipeline()

    if results is not None:
        print("\nPlotting Results...")
        plot_pipeline_results(results, show=True, save_path="
    gnn_results.png")
```

Listing 9.3: GNN Architecture Code

Bibliography

[1] G. Bergers and S.-M. Fendt. The metabolism of cancer cells during metastasis. *Nature Reviews Cancer*, 21(3):162–180, 2021.

[2] Sara E. Cross, Yu-Sheng Jin, Jianyu Rao, and James K. Gimzewski. Nanomechanical analysis of cells from cancer patients. *Nature Nanotechnology*, 2(12):780–783, 2007.

[3] Tae-Hwan Kim et al. Cancer cells become less deformable and more invasive with activation of β-adrenergic signaling. *Journal of Cell Science*, 129(24):4563–4575, 2016.

[4] Arsen Mkrtchyan, Johan Åström, and Mikko Karttunen. A new model for cell division and migration with spontaneous topology changes. *Soft Matter*, 10(24):4332–4339, 2014.

[5] Danh Nguyen, Lei Tao, and Ying Li. Integration of machine learning and coarse-grained molecular simulations for polymer materials: Physical understandings and molecular design. *Frontiers in Chemistry*, 9:820417, 2022.

[6] Michael A. Webb, Nicholas E. Jackson, Phwey S. Gil, and Juan J. de Pablo. Targeted sequence design within the coarse-grained polymer genome. *Science Advances*, 6(43):eabc6216, 2020.

[7] H. Wen et al. Collective motion of cells modeled as ring polymers. *Soft Matter*, 18(6):1228–1238, 2022.

[8] Yue Zhu, P. B. Sunil Kumar, and Mohamed Laradji. Conformational behavior and self-assembly of disjoint semi-flexible ring polymers adsorbed on solid substrates. *Soft Matter*, 17(21):5427–5435, 2021.