Polymer Molecular Dynamics

A Comprehensive Guide for Computational Scientists

From Atomistic Simulations to Continuum Models

Sreekanth Pannala

January 12, 2026

Contents

Part I

Polymer Physics Fundamentals

Chapter 1

Introduction to Polymers

1.1 What Are Polymers?

Polymers are large molecules (macromolecules) composed of repeating structural units called **monomers**, connected by covalent bonds. The word "polymer" derives from Greek: *poly* (many) + *meros* (parts).

Definition 1.1 (Polymer)**.** *A polymer is a substance composed of macromolecules with molecular weights typically ranging from* 10^3 *to* 10^7 *g/mol, built from smaller repeating units through polymerization reactions.*

Computational Perspective: From a simulation standpoint, polymers present unique challenges:

- **Multiple length scales**: From bond lengths (∼1.5 Å) to chain dimensions (∼100 nm)
- **Multiple time scales**: From bond vibrations (∼fs) to chain relaxation (∼ms or longer)
- **Topological constraints**: Chains cannot pass through each other (entanglements)
- **Conformational entropy**: Enormous number of accessible chain configurations

1.2 Polymer Classification

1.2.1 By Structure

Type	Structure	Examples
Linear	$-A-A-A-A-A-$	HDPE, Nylon, PEO
Branched	Main chain with side chains	LDPE, Glycogen
Cross-linked	3D network	Vulcanized rubber, Epoxy
Star	Multiple arms from central point	Star-PS, Dendrimers
Ring	Cyclic chains	Cyclic DNA, Cyclic PS

1.2.2 By Composition

- **Homopolymer**: Single monomer type ($-A-A-A-A-$)
- **Copolymer**: Two or more monomer types
  - Random: $-A-B-A-A-B-A-B-B-$
  - Alternating: $-A-B-A-B-A-B-$

- Block: $-A-A-A-B-B-B-A-A-A-$
- Graft: Main chain of A with side chains of B

1.2.3 By Thermal Behavior

Type	Characteristics	MD Implications
Thermoplastic	Softens on heating, chains slide past each other	Can simulate melt flow, requires long equilibration
Thermoset	Cross-linked network, doesn't melt	Fixed topology, bond-breaking requires reactive MD
Elastomer	Cross-linked but flexible, large reversible strain	Network topology important, rubber elasticity

1.3 Polymer Nomenclature

1.3.1 Chemical Structure Notation

The repeat unit defines the polymer. For polyethylene:

$$\text{PE}: \quad -[\text{CH}_2-\text{CH}_2]_n- \quad \text{or} \quad -[\text{CH}_2]_{2n}- \tag{1.1}$$

Common polyolefins and their structures:

Polymer	Repeat Unit	Atoms per Repeat
Polyethylene (PE)	$-\text{CH}_2\text{–}\text{CH}_2-$	6
Polypropylene (PP)	$-\text{CH}_2\text{–}\text{CH}(\text{CH}_3)-$	9
Polystyrene (PS)	$-\text{CH}_2\text{–}\text{CH}(\text{C}_6\text{H}_5)-$	16
Poly(methyl methacrylate) (PMMA)	$-\text{CH}_2\text{–}\text{C}(\text{CH}_3)(\text{COOCH}_3)-$	15

1.3.2 Tacticity

For polymers with stereocenters (e.g., PP), the arrangement of side groups matters:

- **Isotactic**: All substituents on same side
- **Syndiotactic**: Alternating arrangement
- **Atactic**: Random arrangement

MD Note: Tacticity significantly affects:

- Crystallization tendency (isotactic > syndiotactic > atactic)
- Glass transition temperature
- Chain stiffness (different C_∞ values)

1.4 Molecular Weight Distribution

Unlike small molecules, polymers have a *distribution* of molecular weights.

1.4.1 Average Molecular Weights

Definition 1.2 (Number-Average Molecular Weight)**.**

$$M_{\mathrm{n}} = \frac{\sum_i N_i M_i}{\sum_i N_i} \tag{1.2}$$

where N_i is the number of chains with molecular weight M_i.

Definition 1.3 (Weight-Average Molecular Weight)**.**

$$M_{\mathrm{w}} = \frac{\sum_i N_i M_i^2}{\sum_i N_i M_i} \tag{1.3}$$

Definition 1.4 (Dispersity (Polydispersity Index))**.**

$$Đ = \frac{M_{\mathrm{w}}}{M_{\mathrm{n}}} \geq 1 \tag{1.4}$$

Typical dispersity values:

- Living polymerization: $Đ \approx 1.0$–1.1
- Anionic: $Đ \approx 1.01$–1.05
- Free radical: $Đ \approx 1.5$–2.0
- Ziegler-Natta (PE): $Đ \approx 3$–10
- Condensation: $Đ \approx 2$

1.4.2 Degree of Polymerization

$$N = \frac{M}{M_0} \tag{1.5}$$

where M_0 is the monomer molecular weight.

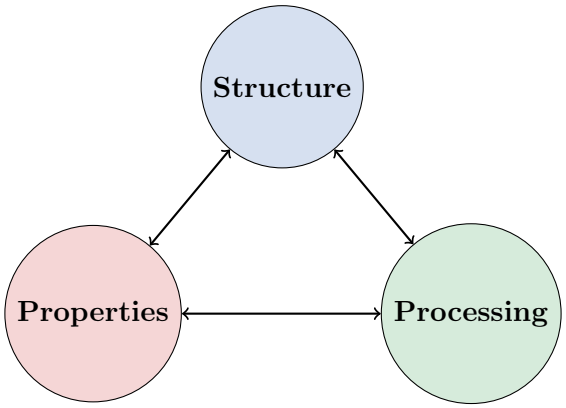
For PE: $M_0 = 28$ g/mol (CH_2–CH_2), so $N = M/28$.

1.4.3 Simulation Considerations

```python
# Molecular weight to chain length conversion
def mw_to_beads(mw, m0=28.0, beads_per_monomer=0.25):
    """
    Convert molecular weight to number of CG beads.

    Args:
        mw: Molecular weight (g/mol)
        m0: Monomer molecular weight (28 for PE)
        beads_per_monomer: CG mapping (0.25 = 4 monomers per bead)

    Returns:
        Number of CG beads
    """
    n_monomers = mw / m0
    n_beads = int(n_monomers * beads_per_monomer)
    return max(n_beads, 2)

# Example: HDPE with Mn = 100,000 g/mol
n_beads = mw_to_beads(100000)  # ~893 beads for 4:1 mapping
```

1.5 Structure-Property-Processing Triangle

The central paradigm of materials science applies to polymers:



1.5.1 Structure

- Chemical composition (monomers, functional groups)
- Architecture (linear, branched, cross-linked)
- Molecular weight and distribution
- Tacticity and stereochemistry
- Crystallinity and morphology

1.5.2 Properties

- Mechanical: modulus, strength, toughness
- Thermal: T_g, T_m, thermal stability
- Rheological: viscosity, elasticity
- Optical: transparency, refractive index
- Transport: permeability, diffusion

1.5.3 Processing

- Melt processing: extrusion, injection molding
- Solution processing: spin coating, fiber spinning
- Solid-state: drawing, annealing
- Additive manufacturing: FDM, SLA

1.6 Length and Time Scales in Polymer Systems

Scale	Length	Time
Bond vibrations	0.01–0.1 Å	10–100 fs
Bond rotation	1–2 Å	1–10 ps
Kuhn segment	10–20 Å	10–100 ps
Radius of gyration	10–1000 Å	ns–μs
Entanglement tube	50–100 Å	μs–ms
Reptation/relaxation	Chain-dependent	ms–hours

Simulation Method Selection:

- **Quantum (DFT)**: Electronic structure, reaction mechanisms (<100 atoms)
- **Atomistic MD**: Local structure, short-time dynamics (10^3–10^6 atoms, ns)
- **Coarse-grained MD**: Chain conformations, entanglements (10^4–10^8 beads, μs)
- **Field theory (SCMFT)**: Phase behavior, morphology (μm, equilibrium)
- **Continuum (FEM)**: Bulk mechanical response (mm–m)

1.7 Key Polymer Physics Concepts Preview

The following chapters will develop these concepts in detail:

1.7.1 Chain Statistics (Chapter 2)

- Random walk models: freely-jointed, freely-rotating, hindered rotation
- Characteristic ratio C_∞ and Kuhn length l_K
- Radius of gyration R_g and end-to-end distance R_ee
- Scaling laws: $R_\mathrm{g} \sim N^\nu$

1.7.2 Thermodynamics (Chapter 3)

- Flory-Huggins theory for polymer solutions/blends
- χ parameter and phase behavior
- UCST/LCST transitions

1.7.3 Rheology (Chapter 4)

- Rouse model (unentangled)
- Reptation theory (entangled)
- Entanglement molecular weight M_e

1.8 Summary for Computational Scientists

Concept	**Computational Relevance**
Repeat unit	Defines force field atom types, building blocks for chain generation
Chain length N	System size, determines computational cost ($\sim N$ to N^3)
Tacticity	Different stereoisomers need different dihedral parameters
Dispersity	May need ensemble of different chain lengths
Architecture	Topology connectivity matrix, branch handling
T_g, T_m	Sets simulation temperature regime

1.9 Exercises

1. Calculate the number of CG beads needed to represent a polystyrene chain with $M_\mathrm{w} = 300,000$ g/mol using a 4:1 mapping (4 monomers per bead).

2. A polymer sample has chains with $N_1 = 100$ chains of $M = 10,000$ g/mol and $N_2 = 50$ chains of $M = 50,000$ g/mol. Calculate M_n, M_w, and dispersity.

3. Explain why atactic polypropylene is amorphous while isotactic polypropylene is semicrystalline. What implications does this have for MD simulation?

4. Estimate the minimum simulation box size needed to contain 10 HDPE chains with $M_\mathrm{w} = 100,000$ g/mol at melt density ($\rho \approx 0.85$ g/cm^3).

1.10 Further Reading

- Rubinstein, M. & Colby, R.H. *Polymer Physics*, Chapter 1
- Painter, P.C. & Coleman, M.M. *Fundamentals of Polymer Science*
- Odian, G. *Principles of Polymerization*

Chapter 2

Polymer Chain Statistics

2.1 The Random Walk Foundation

The statistical description of polymer chains begins with the random walk model—a connection that profoundly links polymer physics to probability theory.

Definition 2.1 (Random Walk)**.** *A random walk is a sequence of steps where each step direction is chosen randomly, independent of previous steps. For a polymer, each "step" represents a bond or segment along the chain backbone.*

2.2 Freely-Jointed Chain (FJC) Model

The simplest polymer model treats the chain as N rigid segments of length b, connected by freely-rotating joints.

2.2.1 Mathematical Description

Each segment vector $\mathbf{r}_i$ has:

- Fixed magnitude: $|\mathbf{r}_i| = b$
- Uniform random direction: $P(\hat{\mathbf{r}}_i) = 1/(4\pi)$
- Independence: $\langle \mathbf{r}_i \cdot \mathbf{r}_j \rangle = b^2 \delta_{ij}$

2.2.2 End-to-End Vector

The end-to-end vector is the sum of all segment vectors:

$$\mathbf{R} = \sum_{i=1}^{N} \mathbf{r}_i \tag{2.1}$$

2.2.3 Mean-Square End-to-End Distance

$$\langle R_{\text{ee}}^2 \rangle = \langle \mathbf{R} \cdot \mathbf{R} \rangle = \langle \sum_{i=1}^{N} \mathbf{r}_i \cdot \sum_{j=1}^{N} \mathbf{r}_j \rangle \tag{2.2}$$

$$= \sum_{i=1}^{N} \sum_{j=1}^{N} \langle \mathbf{r}_i \cdot \mathbf{r}_j \rangle \tag{2.3}$$

$$= \sum_{i=1}^{N} b^2 = Nb^2 \tag{2.4}$$

$$\boxed{\langle R_{\text{ee}}^2 \rangle_{\text{FJC}} = Nb^2} \tag{2.5}$$

This is the fundamental scaling relation: **chain size grows as $\sqrt{N}$.**

2.2.4 Probability Distribution

By the Central Limit Theorem, for large N, the end-to-end vector follows a Gaussian distribution:

$$P(\mathbf{R}) = \left(\frac{3}{2\pi N b^2} \right)^{3/2} \exp\left(-\frac{3R^2}{2Nb^2} \right) \tag{2.6}$$

The probability of finding the chain end at distance R (spherical shell):

$$P(R) = 4\pi R^2 P(\mathbf{R}) = 4\pi R^2 \left(\frac{3}{2\pi N b^2} \right)^{3/2} \exp\left(-\frac{3R^2}{2Nb^2} \right) \tag{2.7}$$

2.3 Freely-Rotating Chain (FRC) Model

Real polymers have fixed bond angles. The FRC model incorporates this constraint.

2.3.1 Fixed Bond Angle

All bond angles are fixed at θ (supplement of the backbone angle):

$$\cos\theta = \frac{\mathbf{r}_i \cdot \mathbf{r}_{i+1}}{b^2} \tag{2.8}$$

For tetrahedral carbon: $\theta = 180 - 109.5 = 70.5$, so $\cos\theta \approx 1/3$.

2.3.2 Bond Vector Correlations

Sequential bond vectors are correlated:

$$\langle \mathbf{r}_i \cdot \mathbf{r}_{i+k} \rangle = b^2 (\cos\theta)^k \tag{2.9}$$

This correlation decays exponentially with separation k.

2.3.3 Mean-Square End-to-End Distance

$$\langle R_{\text{ee}}^2 \rangle = \sum_{i=1}^{N} \sum_{j=1}^{N} \langle \mathbf{r}_i \cdot \mathbf{r}_j \rangle \tag{2.10}$$

$$= Nb^2 + 2 \sum_{i=1}^{N} \sum_{k=1}^{N-i} b^2 (\cos\theta)^k \tag{2.11}$$

$$\approx Nb^2 \frac{1+\cos\theta}{1-\cos\theta} \quad \text{(for large } N\text{)} \tag{2.12}$$

$$\boxed{\langle R_{\text{ee}}^2 \rangle_{\text{FRC}} = Nb^2 \frac{1+\cos\theta}{1-\cos\theta}} \tag{2.13}$$

For tetrahedral angle ($\cos\theta = 1/3$): $\langle R_{\text{ee}}^2 \rangle = 2Nb^2$.

2.4 The Characteristic Ratio

Definition 2.2 (Characteristic Ratio)**.** *The characteristic ratio C_N quantifies chain stiffness by comparing real chain dimensions to the FJC model:*

$$C_N = \frac{\langle R_{\text{ee}}^2 \rangle}{Nb^2} \tag{2.14}$$

In the limit of long chains:

$$C_\infty = \lim_{N \to \infty} C_N \tag{2.15}$$

2.4.1 Physical Interpretation

- $C_\infty = 1$: Ideal FJC (no correlations)
- $C_\infty > 1$: Real chains are stiffer than FJC
- Higher C_∞ = stiffer backbone

2.4.2 Experimental Values

Polymer	C_∞	**Backbone**	**Stiffening Factor**
Polyethylene (PE)	6.7–7.4	C–C–C	Trans/gauche preference
Polypropylene (iPP)	5.5–5.9	C–C–C + CH_3	Methyl interactions
Polystyrene (aPS)	9.5–10.5	C–C–C + phenyl	Bulky side group
PDMS	6.2–6.5	Si–O–Si	Flexible backbone
Cellulose	∼50	Glucose rings	Ring stiffness
DNA	∼100	Double helix	Extreme stiffness

2.5 Kuhn Length and Effective Segments

Definition 2.3 (Kuhn Length)**.** *The Kuhn length l_{K} is the effective segment length that makes a real chain behave like an FJC:*

$$\langle R_{\text{ee}}^2 \rangle = N_K l_{\text{K}}^2 \tag{2.16}$$

where N_K is the number of Kuhn segments.

2.5.1 Relation to Characteristic Ratio

Using contour length conservation: $L = Nb = N_K l_K$

$$l_K = C_\infty b \tag{2.17}$$

$$N_K = \frac{N}{C_\infty} \tag{2.18}$$

2.5.2 Physical Meaning

The Kuhn length represents the distance over which bond orientations become uncorrelated. It is approximately twice the persistence length:

$$l_K \approx 2l_p \tag{2.19}$$

2.6 Persistence Length

Definition 2.4 (Persistence Length)**.** *The persistence length l_p characterizes the decay of tangent-tangent correlations along the chain:*

$$\langle \hat{\mathbf{t}}(0) \cdot \hat{\mathbf{t}}(s) \rangle = \exp(-s/l_p) \tag{2.20}$$

where $\hat{\mathbf{t}}(s)$ is the unit tangent vector at contour position s.

2.6.1 Relation to Bond Correlations

For discrete chains:

$$\langle \cos\theta_{i,i+k} \rangle = \exp(-kb/l_p) \tag{2.21}$$

Inverting:

$$l_p = -\frac{b}{\ln\langle \cos\theta \rangle} \tag{2.22}$$

2.6.2 Worm-Like Chain (WLC) Model

For semiflexible polymers with large l_p:

$$\langle R_{ee}^2 \rangle = 2l_p L \left[1 - \frac{l_p}{L} \left(1 - e^{-L/l_p} \right) \right] \tag{2.23}$$

Limiting cases:

- $L \gg l_p$: $\langle R_{ee}^2 \rangle \approx 2l_p L$ (Gaussian coil)
- $L \ll l_p$: $\langle R_{ee}^2 \rangle \approx L^2$ (rigid rod)

2.7 Radius of Gyration

Definition 2.5 (Radius of Gyration)**.** *The radius of gyration measures the average distance of monomers from the center of mass:*

$$R_g^2 = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{r}_i - \mathbf{r}_{cm})^2 \tag{2.24}$$

2.7.1 Relation to End-to-End Distance

For Gaussian chains (large N):

$$\boxed{\langle R_g^2 \rangle = \frac{\langle R_{ee}^2 \rangle}{6}} \tag{2.25}$$

Or equivalently:

$$\frac{\langle R_{ee}^2 \rangle}{\langle R_g^2 \rangle} = 6 \quad \text{(Gaussian chains)} \tag{2.26}$$

This ratio is a diagnostic for chain ideality in simulations.

2.7.2 Experimental Measurement

R_g is directly measurable via:

- Light scattering (Zimm plot)
- Small-angle X-ray scattering (SAXS)
- Small-angle neutron scattering (SANS)

2.8 Excluded Volume and Real Chains

Ideal chain models ignore self-avoidance. Real chains cannot overlap.

2.8.1 Flory Theory

Flory's mean-field argument balances:

- Elastic energy: $F_{\text{elastic}} \sim k_B T \frac{R^2}{Nb^2}$
- Excluded volume energy: $F_{\text{ev}} \sim k_B T v \frac{N^2}{R^3}$

Minimizing total free energy:

$$R \sim N^{\nu} b, \quad \nu = \frac{3}{d+2} \tag{2.27}$$

For 3D ($d = 3$): $\nu = 3/5 = 0.6$
More accurate renormalization group result: $\nu \approx 0.588$.

2.8.2 Scaling Regimes

Condition	Flory Exponent ν	Scaling
Good solvent (swollen)	0.588	$R_g \sim N^{0.588}$
Theta solvent (ideal)	0.5	$R_g \sim N^{0.5}$
Poor solvent (collapsed)	1/3	$R_g \sim N^{1/3}$
Melt (screened)	0.5	$R_g \sim N^{0.5}$

Key insight: In polymer melts, excluded volume is *screened* by surrounding chains, and ideal chain statistics apply!

2.9 Gyration Tensor and Shape

The full shape information is in the gyration tensor:

$$S_{\alpha\beta} = \frac{1}{N}\sum_{i=1}^{N}(r_{i,\alpha} - r_{\mathrm{cm},\alpha})(r_{i,\beta} - r_{\mathrm{cm},\beta}) \tag{2.28}$$

2.9.1 Eigenvalues and Shape Parameters

Eigenvalues $\lambda_1 \leq \lambda_2 \leq \lambda_3$ give principal dimensions:

$$R_g^2 = \lambda_1 + \lambda_2 + \lambda_3 \tag{2.29}$$

Shape descriptors:

$$\text{Asphericity:} \quad b = \lambda_3 - \frac{1}{2}(\lambda_1 + \lambda_2) \tag{2.30}$$

$$\text{Acylindricity:} \quad c = \lambda_2 - \lambda_1 \tag{2.31}$$

$$\text{Relative anisotropy:} \quad \kappa^2 = \frac{3}{2}\frac{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}{(\lambda_1 + \lambda_2 + \lambda_3)^2} - \frac{1}{2} \tag{2.32}$$

κ^2 ranges from 0 (sphere) to 1 (rod).

2.10 Computational Implementation

2.10.1 Computing C_N from MD

```
def compute_characteristic_ratio(positions: torch.Tensor,
                                 bond_length: float) -> float:
    """
    Compute characteristic ratio from MD trajectory.

    Args:
        positions: Backbone positions (N, 3) on GPU
        bond_length: Nominal bond length b

    Returns:
        C_N = <Ree^2> / (N * b^2)
    """
    # End-to-end vector (GPU)
    R_ee = positions[-1] - positions[0]
    R_ee_sq = (R_ee ** 2).sum().item()

    # Number of bonds
    N = positions.shape[0] - 1

    # Characteristic ratio
    C_N = R_ee_sq / (N * bond_length ** 2)
    return C_N
```

2.10.2 Computing R_g from MD

```
def compute_radius_of_gyration(positions: torch.Tensor) -> float:
    """
    Compute radius of gyration (GPU-optimized).

    Args:
        positions: Atom positions (N, 3)

    Returns:
        Radius of gyration in same units as positions
    """
    # Center of mass
    com = positions.mean(dim=0)

    # Squared distances from COM
    diff = positions - com
    r_sq = (diff ** 2).sum(dim=1)

    # Mean-square distance = Rg^2
    Rg_sq = r_sq.mean()

    return torch.sqrt(Rg_sq).item()
```

2.10.3 Bond Vector Correlation Function

```
def compute_bond_correlations(positions: torch.Tensor,
                              max_sep: int = 50) -> dict:
    """
    Compute bond vector orientation correlations <u_i . u_{i+k}>.

    This measures the decay of chain memory and can be used
    to extract persistence length.

    Args:
        positions: Chain positions (N, 3)
        max_sep: Maximum separation to compute

    Returns:
        Dictionary with separations and correlations
    """
    # Bond vectors (all on GPU)
    bond_vecs = positions[1:] - positions[:-1]

    # Normalize
    norms = torch.norm(bond_vecs, dim=1, keepdim=True)
    unit_vecs = bond_vecs / (norms + 1e-10)

    n_bonds = unit_vecs.shape[0]
    correlations = []

    for k in range(min(max_sep, n_bonds)):
        if k == 0:
            correlations.append(1.0)
        else:
            # Vectorized dot product for separation k
            dots = (unit_vecs[:-k] * unit_vecs[k:]).sum(dim=1)
            correlations.append(dots.mean().item())
```

```
    return {
        'separations': list(range(len(correlations))),
        'correlations': correlations
    }
```

2.11 Mapping Between Models

Quantity	Atomistic	CG	Relation
Bond length	$b_{\mathrm{CC}} = 1.54$ Å	$b_{\mathrm{CG}} \sim 4$ Å	$b_{\mathrm{CG}} = m \cdot b_{\mathrm{CC}}$
Chain length	N_{atoms}	N_{CG}	$N_{\mathrm{CG}} = N_{\mathrm{atoms}}/m$
C_∞	C_∞^{exp}	C_∞^{CG}	Match via κ
R_g	R_g^{exp}	R_g^{CG}	Should match!

The Kuhn length provides the mapping:

$$l_{\mathrm{K}} = C_\infty b_{\mathrm{CC}} = C_\infty^{\mathrm{CG}} b_{\mathrm{CG}} \tag{2.33}$$

2.12 Summary: Key Equations

Property	Equation
FJC end-to-end	$\langle R_{\mathrm{ee}}^2 \rangle = Nb^2$
FRC end-to-end	$\langle R_{\mathrm{ee}}^2 \rangle = Nb^2 \frac{1+\cos\theta}{1-\cos\theta}$
Characteristic ratio	$C_\infty = \langle R_{\mathrm{ee}}^2 \rangle / (Nb^2)$
Kuhn length	$l_{\mathrm{K}} = C_\infty b$
Persistence length	$l_{\mathrm{p}} = -b/\ln\langle\cos\theta\rangle$
R_g to R_{ee}	$\langle R_g^2 \rangle = \langle R_{\mathrm{ee}}^2 \rangle / 6$
Flory scaling	$R_g \sim N^\nu$, $\nu = 0.588$ (good solvent)

2.13 Exercises

1. Derive the FRC result $\langle R_{\mathrm{ee}}^2 \rangle = Nb^2(1+\cos\theta)/(1-\cos\theta)$ starting from $\langle \mathbf{r}_i \cdot \mathbf{r}_j \rangle = b^2(\cos\theta)^{|i-j|}$.

2. For polyethylene with $C_\infty = 7.0$ and $b_{\mathrm{CC}} = 1.54$ Å:

   - Calculate the Kuhn length
   - For a chain with $N = 1000$ bonds, calculate $\langle R_{\mathrm{ee}}^2 \rangle^{1/2}$ and $\langle R_g^2 \rangle^{1/2}$
   - How many Kuhn segments does this chain have?

3. Write a Python function to fit the persistence length from bond correlation data using exponential decay.

4. Explain why $\langle R_{\mathrm{ee}}^2 \rangle / \langle R_g^2 \rangle = 6$ for Gaussian chains. What happens to this ratio for collapsed globules?

2.14 Further Reading

- Rubinstein & Colby, *Polymer Physics*, Chapters 2–3
- Doi & Edwards, *Theory of Polymer Dynamics*, Chapter 2
- Flory, *Statistical Mechanics of Chain Molecules*
- de Gennes, *Scaling Concepts in Polymer Physics*

Chapter 3

Thermodynamics and Phase Behavior

3.1 Introduction to Polymer Thermodynamics

The thermodynamics of polymer systems is fundamentally different from that of small molecules due to the enormous size disparity between polymer chains and solvent molecules. This chapter develops the theoretical framework for understanding polymer mixing, phase separation, and the computational approaches to study these phenomena.

Definition 3.1 (Polymer Solution)**.** *A polymer solution is a mixture of polymer chains dissolved in a low molecular weight solvent. The thermodynamic properties depend critically on the polymer-solvent interactions characterized by the Flory-Huggins interaction parameter χ.*

3.2 Flory-Huggins Theory

3.2.1 Lattice Model Foundation

The Flory-Huggins theory provides a mean-field description of polymer mixing using a lattice model. Consider a lattice with n_0 total sites:

- n_1 solvent molecules, each occupying 1 site
- n_2 polymer chains, each with N segments occupying N sites
- Volume fractions: $\phi_1 = n_1/n_0$ (solvent), $\phi_2 = Nn_2/n_0$ (polymer)

The constraint is:

$$n_1 + Nn_2 = n_0 \quad \Rightarrow \quad \phi_1 + \phi_2 = 1 \tag{3.1}$$

3.2.2 Entropy of Mixing

The combinatorial entropy of mixing per lattice site:

$$\boxed{\frac{\Delta S_{\text{mix}}}{k_{\text{B}} n_0} = -\frac{\phi_1}{1}\ln\phi_1 - \frac{\phi_2}{N}\ln\phi_2} \tag{3.2}$$

For a polymer blend with chains of length N_A and N_B:

$$\frac{\Delta S_{\text{mix}}}{k_{\text{B}} n_0} = -\frac{\phi_A}{N_A}\ln\phi_A - \frac{\phi_B}{N_B}\ln\phi_B \tag{3.3}$$

Key insight: For high molecular weight polymers ($N \to \infty$), the entropy of mixing approaches zero, making the enthalpy term dominant.

3.2.3 Enthalpy of Mixing

The interaction enthalpy in mean-field approximation:

$$\Delta H_{\text{mix}} = k_{\text{B}} T \chi n_0 \phi_1 \phi_2 \tag{3.4}$$

where the Flory-Huggins parameter χ is defined as:

$$\boxed{\chi = \frac{z}{2k_{\text{B}}T}\left[2\epsilon_{12} - \epsilon_{11} - \epsilon_{22}\right]} \tag{3.5}$$

Here z is the coordination number and ϵ_{ij} are pairwise interaction energies.

3.2.4 Free Energy of Mixing

The total Gibbs free energy of mixing per site:

$$\boxed{\frac{\Delta G_{\text{mix}}}{k_{\text{B}}T n_0} = \frac{\phi_1}{1}\ln\phi_1 + \frac{\phi_2}{N}\ln\phi_2 + \chi\phi_1\phi_2} \tag{3.6}$$

For polymer blends:

$$\frac{\Delta G_{\text{mix}}}{k_{\text{B}}T n_0} = \frac{\phi_A}{N_A}\ln\phi_A + \frac{\phi_B}{N_B}\ln\phi_B + \chi_{AB}\phi_A\phi_B \tag{3.7}$$

3.3 The Chi Parameter

3.3.1 Temperature Dependence

The χ parameter typically has both enthalpic and entropic contributions:

$$\chi = \chi_H + \chi_S = \frac{A}{T} + B \tag{3.8}$$

where A and B are empirical constants determined from experiment or simulation.

3.3.2 Measurement from MD Simulation

The χ parameter can be computed from simulations using several methods:

```
def compute_chi_from_coordination(traj, type_A, type_B, cutoff=6.0):
    """
    Compute chi parameter from coordination number analysis.

    chi = (z/2) * (2*p_AB - p_AA - p_BB) / kT

    where p_ij is the pair interaction probability.
    """
    import numpy as np

    n_frames = len(traj)
    z_AA, z_AB, z_BB = 0.0, 0.0, 0.0

    for frame in traj:
        positions = frame.positions
        types = frame.types

        # Count neighbors within cutoff
```

```
        for i, pos_i in enumerate(positions):
            for j, pos_j in enumerate(positions):
                if i >= j:
                    continue
                r = np.linalg.norm(pos_i - pos_j)
                if r < cutoff:
                    if types[i] == type_A and types[j] == type_A:
                        z_AA += 2
                    elif types[i] == type_B and types[j] == type_B:
                        z_BB += 2
                    else:
                        z_AB += 2

    # Normalize and compute chi
    n_A = sum(1 for t in traj[0].types if t == type_A)
    n_B = sum(1 for t in traj[0].types if t == type_B)

    p_AB = z_AB / (n_A + n_B) / n_frames
    p_AA = z_AA / n_A / n_frames
    p_BB = z_BB / n_B / n_frames

    z = (z_AA + z_AB + z_BB) / (n_A + n_B) / n_frames
    chi = 0.5 * z * (2*p_AB - p_AA - p_BB)

    return chi
```

3.3.3 Alternative Methods for Chi Calculation

Method 1: Energy-based approach

$$\chi = \frac{\langle U_{AB} \rangle - \frac{1}{2}(\langle U_{AA} \rangle + \langle U_{BB} \rangle)}{k_B T} \tag{3.9}$$

Method 2: Structure factor extrapolation

$$\lim_{q \to 0} S^{-1}(q) = \frac{1}{N_A \phi_A} + \frac{1}{N_B \phi_B} - 2\chi \tag{3.10}$$

3.4 Phase Diagrams

3.4.1 Spinodal and Binodal

The phase diagram is determined by the conditions:

Binodal (coexistence curve): Equal chemical potentials

$$\mu_1(\phi') = \mu_1(\phi'') \quad \text{and} \quad \mu_2(\phi') = \mu_2(\phi'') \tag{3.11}$$

Spinodal (stability limit):

$$\frac{\partial^2 \Delta G_{\text{mix}}}{\partial \phi^2} = 0 \tag{3.12}$$

For polymer-solvent systems:

$$\boxed{\chi_{\text{spinodal}} = \frac{1}{2}\left(\frac{1}{\phi} + \frac{1}{N(1-\phi)}\right)} \tag{3.13}$$

3.4.2 Critical Point

The critical point occurs where:

$$\frac{\partial^2 \Delta G}{\partial \phi^2} = 0 \quad \text{and} \quad \frac{\partial^3 \Delta G}{\partial \phi^3} = 0 \tag{3.14}$$

For polymer-solvent ($N_1 = 1$, $N_2 = N$):

$$\phi_c = \frac{1}{1 + \sqrt{N}} \approx \frac{1}{\sqrt{N}} \quad (N \gg 1) \tag{3.15}$$

$$\chi_c = \frac{1}{2}\left(1 + \frac{1}{\sqrt{N}}\right)^2 \approx \frac{1}{2} + \frac{1}{\sqrt{N}} \tag{3.16}$$

For symmetric polymer blends ($N_A = N_B = N$):

$$\phi_c = 0.5 \quad \text{and} \quad \chi_c N = 2 \tag{3.17}$$

3.5 UCST and LCST Behavior

3.5.1 Upper Critical Solution Temperature (UCST)

UCST behavior occurs when χ decreases with temperature (typical case):

$$\chi = \frac{A}{T} + B \quad \text{with } A > 0 \tag{3.18}$$

- System is miscible at high T (low χ)
- Phase separation occurs upon cooling
- Examples: PS/cyclohexane, PIB/diisobutyl ketone

3.5.2 Lower Critical Solution Temperature (LCST)

LCST behavior requires χ to increase with temperature:

$$\chi = \frac{A}{T} + B \quad \text{with } A < 0 \text{ or compressibility effects} \tag{3.19}$$

- System is miscible at low T
- Phase separation occurs upon heating
- Driven by entropic effects (free volume, hydrogen bonding)
- Examples: PEO/water, PNIPAM/water

```python
def compute_phase_diagram(N, chi_range, n_points=100):
    """
    Compute spinodal and binodal for Flory-Huggins theory.
    """
    import numpy as np
    from scipy.optimize import brentq

    phi = np.linspace(0.01, 0.99, n_points)

    def free_energy(phi, chi, N):
```

```
        return (phi * np.log(phi) + (1-phi)/N * np.log(1-phi)
                + chi * phi * (1-phi))

    def d2G_dphi2(phi, chi, N):
        return 1/phi + 1/(N*(1-phi)) - 2*chi

    spinodal = []
    for chi in chi_range:
        # Find spinodal points
        try:
            phi1 = brentq(lambda p: d2G_dphi2(p, chi, N), 0.01, 0.5)
            phi2 = brentq(lambda p: d2G_dphi2(p, chi, N), 0.5, 0.99)
            spinodal.append((chi, phi1, phi2))
        except ValueError:
            pass

    return np.array(spinodal)
```

3.6 Polymer Blends

3.6.1 Miscibility Criteria

For two polymers to be miscible:

$$\chi_{AB} < \chi_c = \frac{1}{2}\left(\frac{1}{\sqrt{N_A}} + \frac{1}{\sqrt{N_B}}\right)^2 \tag{3.20}$$

For high molecular weight polymers ($N_A, N_B \gg 1$):

$$\chi_c \approx 0 \quad \Rightarrow \quad \text{Most polymer blends are immiscible} \tag{3.21}$$

3.6.2 Compatibilization

Strategies to improve blend miscibility:

- Block copolymer addition (reduces interfacial tension)
- Reactive compatibilization
- Specific interactions (H-bonding, ionic)

3.7 Solubility Parameters

3.7.1 Hildebrand Solubility Parameter

The solubility parameter connects χ to molecular cohesive energy:

$$\delta = \sqrt{\frac{\Delta U_{\text{vap}}}{V_m}} = \sqrt{\frac{E_{\text{coh}}}{V_m}} \tag{3.22}$$

The χ parameter can be estimated from:

$$\chi \approx \frac{V_r}{k_{\text{B}}T}(\delta_1 - \delta_2)^2 \tag{3.23}$$

where V_r is a reference volume (typically monomer volume).

3.7.2 Computation from MD

```
def compute_solubility_parameter(traj, box_volume):
    """
    Compute Hildebrand solubility parameter from MD.

    delta = sqrt(E_coh / V)

    where E_coh = E_intra - E_inter (per molecule)
    """
    import numpy as np

    # Compute cohesive energy density
    E_total = []
    E_bonded = []

    for frame in traj:
        E_total.append(frame.potential_energy)
        E_bonded.append(frame.bonded_energy)

    # Cohesive energy is nonbonded (intermolecular) contribution
    E_nonbonded = np.mean(E_total) - np.mean(E_bonded)

    # Convert to J/cm^3 and compute delta
    # E_coh in kJ/mol, V in nm^3
    CED = -E_nonbonded / box_volume  # kJ/mol/nm^3
    CED_SI = CED * 1e6 / 6.022e23    # J/m^3

    delta = np.sqrt(CED_SI) / 1e3    # MPa^0.5

    return delta
```

3.8 Phase Separation Kinetics

3.8.1 Spinodal Decomposition

In the unstable region (inside spinodal), phase separation proceeds via spinodal decomposition characterized by:

$$\frac{\partial \phi}{\partial t} = M \nabla^2 \left(\frac{\partial f}{\partial \phi} - \kappa \nabla^2 \phi \right) \tag{3.24}$$

This is the Cahn-Hilliard equation where M is mobility and κ is the gradient energy coefficient.

3.8.2 Nucleation and Growth

In the metastable region (between binodal and spinodal), phase separation requires nucleation:

$$\Delta G^* = \frac{16\pi\gamma^3}{3(\Delta g)^2} \tag{3.25}$$

where γ is interfacial tension and Δg is the bulk free energy difference.

3.9 MD Simulation of Phase Behavior

3.9.1 Detecting Phase Separation

```python
def analyze_phase_separation(positions, types, n_bins=20):
    """
    Analyze phase separation using local composition analysis.
    """
    import numpy as np

    # Divide box into cells
    box = np.max(positions, axis=0) - np.min(positions, axis=0)
    cell_size = box / n_bins

    # Count A and B in each cell
    local_phi = np.zeros((n_bins, n_bins, n_bins))

    for pos, t in zip(positions, types):
        i, j, k = (pos / cell_size).astype(int) % n_bins
        if t == 'A':
            local_phi[i,j,k] += 1

    # Normalize
    total_counts = np.histogram(
        positions, bins=n_bins, range=[(0, b) for b in box]
    )[0]
    local_phi /= (total_counts + 1e-10)

    # Phase separation indicated by bimodal distribution
    phi_flat = local_phi.flatten()
    phi_flat = phi_flat[phi_flat > 0]

    return np.std(phi_flat)  # High std = phase separated
```

3.9.2 Structure Factor Analysis

The collective structure factor reveals phase separation:

$$S(q) = \frac{1}{N}\langle \rho_q \rho_{-q} \rangle \tag{3.26}$$

Peak at low q indicates domain formation with characteristic size $d \sim 2\pi/q^*$.

3.10 Theta Solvent and Excluded Volume

3.10.1 Theta Temperature

At the theta temperature Θ, attractive and repulsive interactions cancel:

$$\chi(\Theta) = \frac{1}{2} \tag{3.27}$$

- $T > \Theta$: Good solvent, $R_g \sim N^{0.588}$ (excluded volume)
- $T = \Theta$: Theta solvent, $R_g \sim N^{0.5}$ (ideal chain)
- $T < \Theta$: Poor solvent, chain collapse

3.10.2 Second Virial Coefficient

The second virial coefficient relates to χ:

$$A_2 = \frac{N_A v_1^2}{M_2^2 V_1}\left(\frac{1}{2} - \chi\right) \tag{3.28}$$

At theta conditions: $A_2 = 0$.

3.11 Summary and Key Equations

Quantity	Expression
Free energy of mixing	$\frac{\Delta G}{k_B T n_0} = \frac{\phi}{N}\ln\phi + (1-\phi)\ln(1-\phi) + \chi\phi(1-\phi)$
Critical χ (blend)	$\chi_c N = 2$ (symmetric)
Critical composition	$\phi_c = 1/(1+\sqrt{N})$
Theta condition	$\chi = 1/2$
Solubility parameter	$\delta = \sqrt{E_{\text{coh}}/V}$

3.12 Exercises

1. Derive the spinodal condition for a symmetric polymer blend ($N_A = N_B = N$).
2. For polystyrene ($N = 1000$) in cyclohexane, $\chi = 0.5 + 100/T$. Calculate the theta temperature and critical temperature.
3. Write an MD simulation protocol to measure the χ parameter between two polymer species using the coordination number method.
4. Explain why LCST behavior is common in aqueous polymer solutions but rare in organic solvents.
5. Design a coarse-grained simulation to study spinodal decomposition in a polymer blend. What system sizes and time scales are needed?

3.13 Further Reading

- Rubinstein, M. & Colby, R.H. *Polymer Physics*, Chapter 4
- Flory, P.J. *Principles of Polymer Chemistry*
- de Gennes, P.G. *Scaling Concepts in Polymer Physics*
- Binder, K. *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*

Chapter 4

Polymer Rheology and Viscoelasticity

4.1 Introduction to Polymer Rheology

Rheology is the study of flow and deformation of matter. Polymers exhibit complex rheological behavior due to their chain-like structure, combining viscous (liquid-like) and elastic (solid-like) responses. This viscoelastic behavior is central to polymer processing and applications.

Definition 4.1 (Viscoelasticity)**.** *Viscoelastic materials exhibit both viscous and elastic characteristics when undergoing deformation. The response depends on the time scale of observation relative to molecular relaxation times.*

4.2 Linear Viscoelasticity

4.2.1 Stress Relaxation

When a polymer is subjected to a step strain γ_0, the stress relaxes over time:

$$\sigma(t) = \gamma_0 G(t) \tag{4.1}$$

where $G(t)$ is the stress relaxation modulus. For polymers:

$$G(t) = \sum_i G_i \exp(-t/\tau_i) \tag{4.2}$$

4.2.2 Creep Compliance

Under constant stress σ_0, the strain evolves as:

$$\gamma(t) = \sigma_0 J(t) \tag{4.3}$$

where $J(t)$ is the creep compliance.

4.2.3 Dynamic Moduli

For oscillatory shear $\gamma(t) = \gamma_0 \sin(\omega t)$:

$$\sigma(t) = \gamma_0 [G'(\omega) \sin(\omega t) + G''(\omega) \cos(\omega t)] \tag{4.4}$$

Storage modulus (elastic response):

$$G'(\omega) = \omega \int_0^\infty G(t) \sin(\omega t)\, \mathrm{d}t \tag{4.5}$$

Loss modulus (viscous response):

$$G''(\omega) = \omega \int_0^\infty G(t) \cos(\omega t)\, \mathrm{d}t \tag{4.6}$$

Complex modulus:

$$\boxed{G^*(\omega) = G'(\omega) + iG''(\omega)} \tag{4.7}$$

4.3 The Rouse Model

4.3.1 Model Description

The Rouse model treats a polymer as N beads connected by Gaussian springs, neglecting hydrodynamic interactions and entanglements.

4.3.2 Equation of Motion

For bead n:

$$\zeta \frac{\mathrm{d}\mathbf{R}_n}{\mathrm{d}t} = k_s(\mathbf{R}_{n+1} + \mathbf{R}_{n-1} - 2\mathbf{R}_n) + \mathbf{f}_n(t) \tag{4.8}$$

where:

- ζ = monomeric friction coefficient
- $k_s = 3k_\mathrm{B}T/b^2$ = spring constant
- $\mathbf{f}_n(t)$ = random Brownian force with $\langle \mathbf{f}_n(t) \cdot \mathbf{f}_m(t') \rangle = 6k_\mathrm{B}T\zeta\delta_{nm}\delta(t-t')$

4.3.3 Normal Modes

The Rouse modes are:

$$\mathbf{X}_p(t) = \frac{1}{N} \sum_{n=1}^{N} \mathbf{R}_n(t) \cos\left(\frac{p\pi(n-1/2)}{N}\right) \tag{4.9}$$

Each mode relaxes independently with relaxation time:

$$\boxed{\tau_p = \frac{\tau_R}{p^2} \quad \text{where} \quad \tau_R = \frac{\zeta N^2 b^2}{3\pi^2 k_\mathrm{B}T}} \tag{4.10}$$

τ_R is the longest Rouse relaxation time.

4.3.4 Rouse Model Predictions

Diffusion coefficient:

$$D = \frac{k_\mathrm{B}T}{N\zeta} \propto N^{-1} \tag{4.11}$$

Zero-shear viscosity:

$$\eta_0 = \frac{\pi^2}{12} \frac{\rho k_\mathrm{B}T}{M_0} \tau_R \propto N \tag{4.12}$$

Stress relaxation modulus:

$$G(t) = \frac{\rho k_\mathrm{B}T}{M_0} \sum_{p=1}^{N-1} \exp(-2t/\tau_p) \tag{4.13}$$

4.3.5 Dynamic Moduli (Rouse)

$$G'(\omega) = \frac{\rho k_B T}{M_0} \sum_{p=1}^{N} \frac{(\omega\tau_p)^2}{1+(\omega\tau_p)^2} \tag{4.14}$$

$$G''(\omega) = \frac{\rho k_B T}{M_0} \sum_{p=1}^{N} \frac{\omega\tau_p}{1+(\omega\tau_p)^2} \tag{4.15}$$

In the intermediate frequency regime:

$$G'(\omega) \sim G''(\omega) \sim \omega^{1/2} \tag{4.16}$$

4.4 Entanglements

4.4.1 The Entanglement Concept

For chains longer than a critical length, topological constraints (entanglements) restrict chain motion:

Definition 4.2 (Entanglement Molecular Weight)**.** *The entanglement molecular weight M_e is the average molecular weight between entanglement points:*

$$M_e = \frac{\rho k_B T}{G_N^0} \tag{4.17}$$

where G_N^0 is the plateau modulus.

Polymer	M_e (g/mol)	G_N^0 (MPa)
Polyethylene	1,250	2.6
Polystyrene	18,100	0.2
PMMA	10,000	0.31
PEO	1,730	1.8
Polyisoprene	6,190	0.35

4.4.2 Number of Entanglements

$$Z = \frac{M}{M_e} = \frac{N}{N_e} \tag{4.18}$$

For entangled dynamics: $Z > 1$ (typically need $Z > 2$–3 for clear effects).

4.5 Reptation Theory

4.5.1 The Tube Model

De Gennes proposed that entangled chains are confined to a "tube" formed by surrounding chains. Chain motion occurs by reptation—snake-like diffusion along the tube contour.

4.5.2 Tube Parameters

Tube diameter:

$$a = \sqrt{N_\mathrm{e}}\, b \tag{4.19}$$

Tube length (primitive path):

$$L = \frac{Nb^2}{a} = \frac{N}{\sqrt{N_\mathrm{e}}}\, b = Z\sqrt{N_\mathrm{e}}\, b \tag{4.20}$$

4.5.3 Reptation Time

The time for the chain to completely escape its original tube:

$$\boxed{\tau_d = \frac{L^2}{D_c} = \frac{\zeta N^3 b^2}{\pi^2 k_\mathrm{B} T N_\mathrm{e}} = \tau_R \frac{N}{N_\mathrm{e}} = 3Z^3 \tau_e} \tag{4.21}$$

where $\tau_e = \tau_R / Z^2$ is the entanglement time.

4.5.4 Reptation Predictions

Diffusion coefficient:

$$D \propto N^{-2} \quad \text{(cf. } D \propto N^{-1} \text{ for Rouse)} \tag{4.22}$$

Zero-shear viscosity:

$$\eta_0 \propto N^3 \quad \text{(experiment: } \eta_0 \propto N^{3.4}\text{)} \tag{4.23}$$

The discrepancy (N^3 vs $N^{3.4}$) is attributed to:

- Contour length fluctuations
- Constraint release
- Tube dilation

4.6 Stress Relaxation in Entangled Melts

4.6.1 Relaxation Regimes

Time Scale	Regime	$G(t)$	Physics
$t < \tau_e$	Rouse	$\sim t^{-1/2}$	Free Rouse motion
$\tau_e < t < \tau_R$	Transition	$\sim t^{-1/4}$	Constrained Rouse
$\tau_R < t < \tau_d$	Plateau	$\approx G_N^0$	Entanglement network
$t > \tau_d$	Terminal	$\sim \exp(-t/\tau_d)$	Reptation

4.6.2 Plateau Modulus

In the plateau region:

$$\boxed{G_N^0 = \frac{\rho k_\mathrm{B} T}{M_e} = \frac{4}{5}\frac{\rho k_\mathrm{B} T}{N_\mathrm{e} M_0}} \tag{4.24}$$

4.7 Computing Viscoelastic Properties from MD

4.7.1 Green-Kubo Formula for Viscosity

The shear viscosity is computed from stress fluctuations:

$$\eta = \frac{V}{k_B T} \int_0^\infty \langle \sigma_{\alpha\beta}(0) \sigma_{\alpha\beta}(t) \rangle \, dt \tag{4.25}$$

```python
def compute_viscosity_gk(stress_tensor, dt, temperature, volume):
    """
    Compute viscosity using Green-Kubo relation.

    Args:
        stress_tensor: Array of shape (n_frames, 3, 3)
        dt: Time step between frames
        temperature: Temperature in K
        volume: System volume in nm^3

    Returns:
        Viscosity in Pa.s
    """
    import numpy as np
    from scipy import integrate

    kB = 1.380649e-23  # J/K

    # Extract off-diagonal components (xy, xz, yz)
    sxy = stress_tensor[:, 0, 1]
    sxz = stress_tensor[:, 0, 2]
    syz = stress_tensor[:, 1, 2]

    # Compute autocorrelation functions
    def autocorr(x):
        n = len(x)
        result = np.correlate(x - np.mean(x), x - np.mean(x), mode='full')
        return result[n-1:] / (n - np.arange(n))

    acf_xy = autocorr(sxy)
    acf_xz = autocorr(sxz)
    acf_yz = autocorr(syz)

    # Average over components
    acf_avg = (acf_xy + acf_xz + acf_yz) / 3

    # Integrate (with cutoff at first zero crossing)
    zero_idx = np.where(acf_avg < 0)[0]
    if len(zero_idx) > 0:
        acf_avg = acf_avg[:zero_idx[0]]

    time = np.arange(len(acf_avg)) * dt
    integral = integrate.trapezoid(acf_avg, time)

    # Convert units: nm^3, kJ/mol/nm^3 -> Pa.s
    volume_m3 = volume * 1e-27
    integral_Pa2s = integral * (1e9)**2  # kPa to Pa

    eta = volume_m3 / (kB * temperature) * integral_Pa2s
```

```
    return eta
```

4.7.2 Stress Relaxation Modulus from MD

```
def compute_relaxation_modulus(stress_tensor, dt, temperature, volume):
    """
    Compute stress relaxation modulus G(t) from equilibrium MD.

    G(t) = (V/kT) * <sigma_ab(0) sigma_ab(t)>
    """
    import numpy as np

    kB = 1.380649e-23

    # Off-diagonal stress components
    sigma_off = []
    for i, j in [(0,1), (0,2), (1,2)]:
        sigma_off.append(stress_tensor[:, i, j])

    # Compute G(t) for each component
    G_t_components = []
    for sigma in sigma_off:
        n = len(sigma)
        G_t = np.zeros(n // 2)
        for tau in range(len(G_t)):
            G_t[tau] = np.mean(sigma[:n-tau] * sigma[tau:n])
        G_t_components.append(G_t)

    # Average and convert units
    G_t = np.mean(G_t_components, axis=0)

    # V/kT factor
    prefactor = (volume * 1e-27) / (kB * temperature)
    G_t *= prefactor * 1e6  # Convert to Pa

    time = np.arange(len(G_t)) * dt

    return time, G_t
```

4.7.3 Non-Equilibrium MD (NEMD)

For direct viscosity measurement under shear:

```
# LAMMPS input for NEMD shear simulation
"""
# Apply constant shear rate
fix shear all deform 1 xy erate 0.001 remap v

# Compute stress tensor
compute stress all pressure NULL

# Output shear stress
variable sxy equal c_stress[4]
fix avgstress all ave/time 100 10 1000 v_sxy file stress.dat
```

```
# Viscosity = -<sigma_xy> / shear_rate
"""
```

4.8 Time-Temperature Superposition

4.8.1 WLF Equation

Viscoelastic data at different temperatures can be superposed using shift factors:

$$\log a_T = \frac{-C_1(T - T_{\text{ref}})}{C_2 + (T - T_{\text{ref}})} \tag{4.26}$$

Universal constants (with $T_{\text{ref}} = T_g$): $C_1 \approx 17.4$, $C_2 \approx 51.6$ K.

4.8.2 Master Curve Construction

```
def construct_master_curve(frequencies, G_data, temperatures, T_ref):
    """
    Construct master curve using time-temperature superposition.

    Args:
        frequencies: List of frequency arrays at each T
        G_data: List of (G', G'') tuples at each T
        temperatures: List of temperatures
        T_ref: Reference temperature

    Returns:
        Shifted frequencies and moduli
    """
    import numpy as np

    # WLF constants
    C1, C2 = 17.4, 51.6

    master_freq = []
    master_Gp = []
    master_Gpp = []

    for freq, (Gp, Gpp), T in zip(frequencies, G_data, temperatures):
        # Compute shift factor
        log_aT = -C1 * (T - T_ref) / (C2 + T - T_ref)
        aT = 10**log_aT

        # Shift frequency
        shifted_freq = freq * aT

        master_freq.extend(shifted_freq)
        master_Gp.extend(Gp)
        master_Gpp.extend(Gpp)

    return np.array(master_freq), np.array(master_Gp), np.array(master_Gpp)
```

4.9 Nonlinear Rheology

4.9.1 Shear Thinning

At high shear rates, polymer melts exhibit shear thinning:

$$\eta(\dot{\gamma}) = \eta_0 \left[1 + (\lambda\dot{\gamma})^2\right]^{(n-1)/2} \tag{4.27}$$

This is the Carreau-Yasuda model with relaxation time λ and power-law index $n < 1$.

4.9.2 Normal Stress Differences

Under shear, polymers develop normal stress differences:

$$N_1 = \sigma_{xx} - \sigma_{yy} > 0 \quad \text{(first normal stress difference)} \tag{4.28}$$

$$N_2 = \sigma_{yy} - \sigma_{zz} < 0 \quad \text{(second normal stress difference)} \tag{4.29}$$

N_1 is responsible for phenomena like rod-climbing (Weissenberg effect).

4.9.3 Extensional Viscosity

For uniaxial extension:

$$\eta_E = \frac{\sigma_{xx} - \sigma_{yy}}{\dot{\epsilon}} \tag{4.30}$$

The Trouton ratio for Newtonian fluids:

$$\text{Tr} = \frac{\eta_E}{\eta} = 3 \tag{4.31}$$

For polymers, $\text{Tr} \gg 3$ due to chain stretching (strain hardening).

4.10 Measuring Rheological Properties in MD

4.10.1 Chain Relaxation Times

```python
def compute_rouse_modes(positions, n_modes=10):
    """
    Compute Rouse mode autocorrelation functions.

    X_p(t) = (1/N) * sum_n R_n(t) * cos(p*pi*(n-0.5)/N)
    """
    import numpy as np

    n_frames, n_chains, n_beads, _ = positions.shape

    # Compute Rouse modes for each chain
    rouse_modes = np.zeros((n_frames, n_chains, n_modes, 3))

    for p in range(1, n_modes + 1):
        for n in range(n_beads):
            coeff = np.cos(p * np.pi * (n + 0.5) / n_beads)
            rouse_modes[:, :, p-1, :] += positions[:, :, n, :] * coeff
        rouse_modes[:, :, p-1, :] /= n_beads

    # Compute autocorrelation for each mode
    tau_p = []
```

```
    for p in range(n_modes):
        mode_acf = []
        for chain in range(n_chains):
            Xp = rouse_modes[:, chain, p, :]
            Xp_sq = np.sum(Xp**2, axis=1)
            acf = np.correlate(Xp_sq, Xp_sq, mode='full')
            acf = acf[len(acf)//2:]
            acf /= acf[0]
            mode_acf.append(acf)

        # Average and fit exponential
        avg_acf = np.mean(mode_acf, axis=0)
        # Find where ACF drops to 1/e
        idx = np.where(avg_acf < 1/np.e)[0]
        tau_p.append(idx[0] if len(idx) > 0 else len(avg_acf))

    return np.array(tau_p)
```

4.10.2 End-to-End Vector Autocorrelation

The longest relaxation time can be estimated from:

$$C(t) = \frac{\langle \mathbf{R}_{ee}(0) \cdot \mathbf{R}_{ee}(t) \rangle}{\langle R_{ee}^2 \rangle} \tag{4.32}$$

4.11 Summary of Scaling Relations

Property	Rouse (unentangled)	Reptation (entangled)
Diffusion D	N^{-1}	N^{-2}
Viscosity η	N^{1}	$N^{3.4}$
Longest τ	N^{2}	N^{3}
$G'(\omega)$, $G''(\omega)$	$\omega^{1/2}$	plateau then terminal

4.12 Exercises

1. Derive the Rouse relaxation spectrum from the normal mode analysis.

2. For a polymer melt with $M = 100,000$ g/mol and $M_e = 5,000$ g/mol, estimate the number of entanglements and the ratio τ_d/τ_R.

3. Implement a Green-Kubo viscosity calculation for your MD trajectory. How long must the simulation run to converge?

4. Design an NEMD simulation to measure shear viscosity. What shear rates are appropriate for polymer melts?

5. Using MD data, construct a master curve for $G'(\omega)$ and $G''(\omega)$. Verify the WLF parameters.

4.13 Further Reading

- Doi, M. & Edwards, S.F. *The Theory of Polymer Dynamics*

- Rubinstein, M. & Colby, R.H. *Polymer Physics*, Chapters 7–9
- Ferry, J.D. *Viscoelastic Properties of Polymers*
- de Gennes, P.G. *Scaling Concepts in Polymer Physics*

Part II

Molecular Simulation Methods

Chapter 5

Molecular Dynamics Fundamentals

5.1 Introduction to Molecular Dynamics

Molecular Dynamics (MD) is a computational method that simulates the time evolution of a system of interacting particles by numerically integrating Newton's equations of motion.

Definition 5.1 (Molecular Dynamics)**.** *A deterministic simulation method that computes particle trajectories* $\{\mathbf{r}_i(t), \mathbf{v}_i(t)\}$ *by solving:*

$$m_i \frac{\mathrm{d}^2 \mathbf{r}_i}{\mathrm{d}t^2} = \mathbf{F}_i = -\boldsymbol{\nabla}_i U(\{\mathbf{r}\}) \tag{5.1}$$

where U *is the potential energy function.*

5.1.1 Why MD for Polymers?

- **Natural dynamics**: Captures relaxation, diffusion, viscosity
- **Configurational sampling**: Explores conformational space
- **Non-equilibrium**: Can apply deformation, flow
- **Scale bridging**: Connects molecular structure to properties

5.2 Equations of Motion

5.2.1 Hamiltonian Formulation

The total energy (Hamiltonian) of the system:

$$H = K + U = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m_i} + U(\{\mathbf{r}\}) \tag{5.2}$$

Hamilton's equations:

$$\dot{\mathbf{r}}_i = \frac{\partial H}{\partial \mathbf{p}_i} = \frac{\mathbf{p}_i}{m_i} \tag{5.3}$$

$$\dot{\mathbf{p}}_i = -\frac{\partial H}{\partial \mathbf{r}_i} = \mathbf{F}_i \tag{5.4}$$

5.2.2 Conservation Laws

In the microcanonical (NVE) ensemble:

- **Energy**: H = constant (symplectic integrator required)
- **Momentum**: $\sum_i \mathbf{p}_i$ = constant (if no external forces)
- **Angular momentum**: $\sum_i \mathbf{r}_i \times \mathbf{p}_i$ = constant (if central forces)

5.3 Numerical Integration

5.3.1 Velocity Verlet Algorithm

The most widely used integrator for MD:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{\mathbf{F}(t)}{2m}\Delta t^2 \tag{5.5}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{\mathbf{F}(t) + \mathbf{F}(t + \Delta t)}{2m}\Delta t \tag{5.6}$$

Properties:

- Time-reversible
- Symplectic (preserves phase space volume)
- $O(\Delta t^2)$ local error, $O(\Delta t^2)$ global error
- Energy drift $\sim \Delta t^2$ per step

```
def velocity_verlet_step(positions: torch.Tensor,
                         velocities: torch.Tensor,
                         forces: torch.Tensor,
                         mass: float,
                         dt: float,
                         compute_forces_fn) -> tuple:
    """
    Single velocity Verlet integration step.

    Args:
        positions: Current positions (N, 3)
        velocities: Current velocities (N, 3)
        forces: Current forces (N, 3)
        mass: Particle mass
        dt: Timestep
        compute_forces_fn: Function to compute forces

    Returns:
        Tuple of (new_positions, new_velocities, new_forces)
    """
    # Half-step velocity update
    v_half = velocities + 0.5 * forces / mass * dt

    # Full-step position update
    positions_new = positions + v_half * dt

    # Compute new forces
```

```
    forces_new = compute_forces_fn(positions_new)

    # Complete velocity update
    velocities_new = v_half + 0.5 * forces_new / mass * dt

    return positions_new, velocities_new, forces_new
```

5.3.2 Timestep Selection

The timestep must resolve the fastest motion in the system:

$$\Delta t \lesssim \frac{1}{10} \times \frac{2\pi}{\omega_{\max}} \tag{5.7}$$

Motion	Period	Typical Δt
C–H stretch	∼10 fs	0.5–1 fs (atomistic)
C–C stretch	∼20 fs	1–2 fs
Angle bend	∼50 fs	2–5 fs
CG (no H)	∼100 fs	5–20 fs

5.4 Temperature and Pressure Control

5.4.1 Instantaneous Temperature

From the equipartition theorem:

$$\langle K \rangle = \frac{f}{2} k_{\mathrm{B}} T \tag{5.8}$$

where $f = 3N - 3$ is the number of degrees of freedom (subtracting COM motion). Instantaneous temperature:

$$T = \frac{2K}{f k_{\mathrm{B}}} = \frac{1}{f k_{\mathrm{B}}} \sum_{i=1}^{N} m_i v_i^2 \tag{5.9}$$

```
def compute_temperature(velocities: torch.Tensor,
                        mass: float,
                        kB: float = 0.001987204) -> float:
    """
    Compute instantaneous temperature.

    Args:
        velocities: Velocities (N, 3) in Angstrom/ps
        mass: Particle mass (g/mol)
        kB: Boltzmann constant (kcal/mol/K)

    Returns:
        Temperature in Kelvin
    """
    # Kinetic energy in reduced units
    # Need unit conversion for velocities in Angstrom/ps
    conv = 20.455**2  # (Angstrom/ps)^2 to (kcal/mol)/(g/mol)

    KE = 0.5 * mass * (velocities ** 2).sum().item() / conv
    N = velocities.shape[0]
    dof = 3 * N - 3  # Degrees of freedom
```

```
    T = 2 * KE / (dof * kB)
    return T
```

5.4.2 Thermostats

Velocity Rescaling

Simple but discontinuous:

$$\mathbf{v}_i^{\text{new}} = \mathbf{v}_i \sqrt{\frac{T_{\text{target}}}{T_{\text{current}}}} \tag{5.10}$$

Berendsen Thermostat

Weak coupling to heat bath:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{T_{\text{target}} - T}{\tau_T} \tag{5.11}$$

Velocity scaling factor:

$$\lambda = \sqrt{1 + \frac{\Delta t}{\tau_T}\left(\frac{T_{\text{target}}}{T} - 1\right)} \tag{5.12}$$

Warning: Does not sample canonical ensemble correctly!

Langevin Thermostat

Adds friction and random noise (Ornstein-Uhlenbeck process):

$$m\dot{\mathbf{v}} = \mathbf{F} - \gamma m\mathbf{v} + \sqrt{2\gamma m k_{\mathrm{B}} T}\,\boldsymbol{\eta}(t) \tag{5.13}$$

where $\boldsymbol{\eta}(t)$ is Gaussian white noise: $\langle \eta_\alpha(t)\eta_\beta(t')\rangle = \delta_{\alpha\beta}\delta(t-t')$.
Implementation (velocity update):

$$c_1 = e^{-\gamma \Delta t} \tag{5.14}$$

$$c_2 = \sqrt{\frac{k_{\mathrm{B}}T}{m}(1 - c_1^2)} \tag{5.15}$$

$$\mathbf{v}^{\text{new}} = c_1 \mathbf{v}^{\text{old}} + c_2 \boldsymbol{R} \tag{5.16}$$

where $\boldsymbol{R}$ is a Gaussian random vector.

```
def langevin_thermostat(velocities: torch.Tensor,
                        mass: float,
                        target_temp: float,
                        gamma: float,
                        dt: float,
                        kB: float = 0.001987204) -> torch.Tensor:
    """
    Apply Langevin thermostat.

    Args:
        velocities: Current velocities (N, 3)
        mass: Particle mass
        target_temp: Target temperature (K)
        gamma: Friction coefficient (1/ps)
        dt: Timestep (ps)
        kB: Boltzmann constant
```

```
    Returns:
        Updated velocities
    """
    # Unit conversion
    conv = 20.455  # sqrt(kcal/mol / (g/mol)) -> Angstrom/ps

    # Friction coefficient
    c1 = torch.exp(torch.tensor(-gamma * dt))

    # Noise amplitude
    c2_sq = kB * target_temp / mass * (1 - c1**2)
    c2 = conv * torch.sqrt(torch.tensor(c2_sq))

    # Apply friction
    v_new = c1 * velocities

    # Add noise
    noise = torch.randn_like(velocities) * c2
    v_new = v_new + noise

    # Remove COM velocity
    v_new = v_new - v_new.mean(dim=0)

    return v_new
```

Nosé-Hoover Thermostat

Extended Lagrangian with additional degree of freedom ξ:

$$\dot{\mathbf{r}}_i = \frac{\mathbf{p}_i}{m_i} \tag{5.17}$$

$$\dot{\mathbf{p}}_i = \mathbf{F}_i - \xi \mathbf{p}_i \tag{5.18}$$

$$\dot{\xi} = \frac{1}{Q}\left(\sum_i \frac{p_i^2}{m_i} - f k_\mathrm{B} T\right) \tag{5.19}$$

where Q is the "mass" of the thermostat (controls coupling strength).
Advantage: Generates correct canonical distribution.

5.4.3 Barostats

For NPT simulations, the box size varies to maintain constant pressure.

Berendsen Barostat

$$\frac{\mathrm{d}P}{\mathrm{d}t} = \frac{P_\mathrm{target} - P}{\tau_P} \tag{5.20}$$

Box scaling: $L \to L(1 + \beta \Delta t (P_\mathrm{target} - P)/3\tau_P)^{1/3}$

Parrinello-Rahman

Extended Lagrangian approach for fully flexible cell.

5.5 Periodic Boundary Conditions (PBC)

5.5.1 Minimum Image Convention

For a cubic box of size L:

$$\mathbf{r}_{ij}^{\text{mic}} = \mathbf{r}_j - \mathbf{r}_i - L \cdot \text{round}\left(\frac{\mathbf{r}_j - \mathbf{r}_i}{L}\right) \tag{5.21}$$

```
def minimum_image(r_ij: torch.Tensor, box_size: float) -> torch.Tensor:
    """
    Apply minimum image convention.

    Args:
        r_ij: Displacement vectors (N, 3) or (N, M, 3)
        box_size: Cubic box size

    Returns:
        Wrapped displacement vectors
    """
    return r_ij - box_size * torch.round(r_ij / box_size)
```

5.5.2 Chain Unwrapping

For polymer chains spanning PBC, the minimum image convention gives **wrong** end-to-end distances. Chains must be unwrapped:

```
def unwrap_chain(positions: torch.Tensor,
                 box_size: float) -> torch.Tensor:
    """
    Unwrap a chain that crosses periodic boundaries.

    Progressive unwrapping: each bead relative to previous.

    Args:
        positions: Chain positions (N, 3)
        box_size: Box size

    Returns:
        Unwrapped positions
    """
    unwrapped = positions.clone()

    for i in range(1, len(positions)):
        diff = positions[i] - unwrapped[i-1]
        shift = torch.round(diff / box_size) * box_size
        unwrapped[i] = positions[i] - shift

    return unwrapped
```

5.6 Force Calculation

5.6.1 Non-Bonded Forces

Non-bonded forces dominate computational cost ($O(N^2)$ without optimization).

Lennard-Jones Force

$$\mathbf{F}_{ij}^{\text{LJ}} = \frac{24\varepsilon}{r}\left[2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]\hat{\mathbf{r}}_{ij} \tag{5.22}$$

Cutoff and Shifting

For efficiency, truncate at r_c:

$$U^{\text{cut}}(r) = \begin{cases} U(r) - U(r_c) & r < r_c \\ 0 & r \geq r_c \end{cases} \tag{5.23}$$

For smooth forces, use force-shifted cutoff:

$$U^{\text{fs}}(r) = U(r) - U(r_c) - (r - r_c)\left.\frac{\mathrm{d}U}{\mathrm{d}r}\right|_{r_c} \tag{5.24}$$

5.6.2 Neighbor Lists

Reduce $O(N^2)$ to $O(N)$ by maintaining lists of nearby particles.

Verlet List

Store neighbors within $r_c + r_{\text{skin}}$. Rebuild when any particle moves more than $r_{\text{skin}}/2$.

```
def should_rebuild_neighbors(positions: torch.Tensor,
                             old_positions: torch.Tensor,
                             skin: float) -> bool:
    """
    Check if neighbor list needs rebuilding.

    Args:
        positions: Current positions
        old_positions: Positions when list was built
        skin: Neighbor list skin distance

    Returns:
        True if rebuild needed
    """
    displacement = positions - old_positions
    max_disp = torch.norm(displacement, dim=1).max().item()
    return max_disp > 0.5 * skin
```

Cell Lists

Divide box into cells of size $\geq r_c$. Only check neighboring cells.

5.6.3 Bonded Forces

Bond Stretching

Harmonic: $F = -k(r - r_0)$
FENE: $F = \frac{kr}{1-(r/R_0)^2}$

Angle Bending

The force on each atom in an angle i-j-k (where j is central) involves derivatives of the angle with respect to positions:

$$\mathbf{F}_i = -\frac{\partial U}{\partial \theta}\frac{\partial \theta}{\partial \mathbf{r}_i} \tag{5.25}$$

Dihedral Torsion

Four-body force based on the angle between planes i-j-k and j-k-l.

5.7 Statistical Ensembles

Ensemble	Fixed	Fluctuating	Use Case
NVE (microcanonical)	N, V, E	T, P	Energy conservation check
NVT (canonical)	N, V, T	E, P	Constant temperature
NPT (isothermal-isobaric)	N, P, T	E, V	Experimental conditions
μVT (grand canonical)	μ, V, T	N, E	Adsorption, open systems

5.8 MD Workflow for Polymers

1. **Initial structure**: Generate chains (see Chapter 8)
2. **Energy minimization**: Remove bad contacts
3. **Equilibration**: NVT then NPT
4. **Production**: Collect data
5. **Analysis**: Compute properties

```
def md_workflow(initial_positions, bonds, angles, params,
                T=400.0, P=1.0):
    """
    Standard MD workflow for polymer simulation.
    """
    # 1. Create simulation
    md = CoarseGrainedMD(initial_positions, bonds, angles,
                         params, device='cuda')

    # 2. Energy minimization
    print("Minimizing...")
    md.energy_minimize(n_steps=1000)

    # 3. NVT equilibration
    print("NVT equilibration...")
    md.run_nvt(n_steps=10000, dt=0.005, temperature=T)

    # 4. Production with trajectory averaging
    print("Production run...")
    results = md.run_equilibration(
        n_equil=10000,
        n_prod=50000,
        dt=0.005,
        temperature=T
```

```
    )

    return results
```

5.9 Units and Conversions

Quantity	Common Units	Reduced Units
Length	Å	σ
Energy	kcal/mol or kJ/mol	ε
Mass	g/mol (amu)	m
Time	ps	$\tau = \sigma\sqrt{m/\varepsilon}$
Temperature	K	ε/k_B
Force	kcal/mol/Å	ε/σ

Conversion: $k_\mathrm{B} = 0.001987204$ kcal/(mol·K)

5.10 Summary

Component	Key Points
Integration	Velocity Verlet (symplectic, time-reversible)
Timestep	$\lesssim 1/10$ fastest period
Thermostat	Langevin or Nosé-Hoover for canonical sampling
PBC	Minimum image for forces, unwrap for chain properties
Neighbor list	Essential for $O(N)$ scaling

5.11 Exercises

1. Implement the velocity Verlet algorithm and verify energy conservation for a harmonic oscillator.
2. Show that the Langevin thermostat satisfies the fluctuation-dissipation theorem.
3. Write a function to compute the pressure from the virial: $P = \frac{Nk_\mathrm{B}T}{V} + \frac{1}{3V}\sum_{i<j} \mathbf{r}_{ij} \cdot \mathbf{F}_{ij}$
4. Explain why Berendsen thermostat/barostat do not generate correct canonical/NPT ensembles.

5.12 Further Reading

- Frenkel & Smit, *Understanding Molecular Simulation*, Chapters 3–6
- Allen & Tildesley, *Computer Simulation of Liquids*, Chapters 3–5
- Tuckerman, *Statistical Mechanics: Theory and Molecular Simulation*

Chapter 6

Atomistic Force Fields for Polymers

6.1 Introduction to Force Fields

A force field is a mathematical description of the potential energy surface governing atomic interactions. For molecular dynamics simulations of polymers, the force field determines structural accuracy, dynamic properties, and thermodynamic behavior.

Definition 6.1 (Force Field)**.** *A force field consists of functional forms for potential energy terms and associated parameters that describe intra- and intermolecular interactions between atoms.*

The total potential energy is typically decomposed as:

$$\boxed{U_{\text{total}} = U_{\text{bond}} + U_{\text{angle}} + U_{\text{dihedral}} + U_{\text{LJ}} + U_{\text{elec}}} \tag{6.1}$$

6.2 Bonded Interactions

6.2.1 Bond Stretching

The simplest form is the harmonic potential:

$$U_{\text{bond}} = \frac{1}{2} k_b (r - r_0)^2 \tag{6.2}$$

For more accurate high-energy behavior, the Morse potential is used:

$$U_{\text{bond}}^{\text{Morse}} = D_e \left[1 - e^{-\alpha(r - r_0)}\right]^2 \tag{6.3}$$

where D_e is the dissociation energy and $\alpha = \sqrt{k_b/(2D_e)}$.

Bond Type	**Example**	k_b (kcal/mol/Å^2)	r_0 (Å)
C–C (sp^3)	PE backbone	310	1.526
C–C (sp^2)	PS ring	469	1.400
C–H	alkyl	340	1.090
C=O	carbonyl	570	1.229
C–O	ether	320	1.410

6.2.2 Angle Bending

Harmonic angle potential:

$$U_{\text{angle}} = \frac{1}{2} k_\theta (\theta - \theta_0)^2 \tag{6.4}$$

Some force fields use cosine-based forms:

$$U_{\text{angle}} = k_\theta (1 - \cos(\theta - \theta_0)) \tag{6.5}$$

Angle Type	**Example**	k_θ (kcal/mol/rad^2)	θ_0 (deg)
C–C–C (sp^3)	PE backbone	63	112.7
C–C–H	alkyl	37	110.7
H–C–H	methyl	33	107.8
C–C–C (sp^2)	aromatic	63	120.0

6.2.3 Dihedral (Torsional) Potentials

Dihedrals are critical for polymer conformational behavior. The general form:

$$\boxed{U_{\text{dihedral}} = \sum_{n=1}^{N} \frac{V_n}{2} [1 + \cos(n\phi - \gamma_n)]} \tag{6.6}$$

OPLS form:

$$U_{\text{dihedral}} = \frac{V_1}{2}[1 + \cos\phi] + \frac{V_2}{2}[1 - \cos 2\phi] + \frac{V_3}{2}[1 + \cos 3\phi] + \frac{V_4}{2}[1 - \cos 4\phi] \tag{6.7}$$

Ryckaert-Bellemans form (used in GROMACS):

$$U_{\text{dihedral}} = \sum_{n=0}^{5} C_n (\cos\psi)^n \quad \text{where } \psi = \phi - 180 \tag{6.8}$$

Example 6.1 (Polyethylene Backbone Torsion)**.** *For the C–C–C–C dihedral in polyethylene:*

- *Trans ($\phi = 180$): lowest energy*
- *Gauche$^+$ ($\phi \approx 65$): ∼0.5 kcal/mol higher*
- *Gauche$^-$ ($\phi \approx 295$): ∼0.5 kcal/mol higher*
- *Eclipsed ($\phi = 0, 120, 240$): barriers ∼3 kcal/mol*

6.2.4 Improper Dihedrals

Improper dihedrals maintain planarity (e.g., aromatic rings, sp^2 carbons):

$$U_{\text{improper}} = \frac{1}{2} k_\chi (\chi - \chi_0)^2 \tag{6.9}$$

6.3 Nonbonded Interactions

6.3.1 Lennard-Jones Potential

The standard 12-6 LJ potential:

$$\boxed{U_{\text{LJ}} = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]} \tag{6.10}$$

- ϵ = well depth (energy)
- σ = distance at which $U = 0$
- $r_{\min} = 2^{1/6}\sigma$ = equilibrium distance

Atom Type	ϵ (kcal/mol)	σ (Å)
C (sp^3)	0.066	3.50
C (sp^2)	0.070	3.55
C (aromatic)	0.070	3.55
H (alkyl)	0.030	2.50
O (ether)	0.170	2.96
O (carbonyl)	0.210	2.96
N (amine)	0.170	3.25

6.3.2 Combining Rules

For unlike atom pairs, combining rules are used:

Lorentz-Berthelot (AMBER, CHARMM):

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}, \quad \epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j} \tag{6.11}$$

Geometric mean (OPLS):

$$\sigma_{ij} = \sqrt{\sigma_i \sigma_j}, \quad \epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j} \tag{6.12}$$

6.3.3 Electrostatic Interactions

Coulombic interactions between partial charges:

$$U_{\text{elec}} = \frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}} \tag{6.13}$$

Long-range electrostatics are handled by:

- Particle Mesh Ewald (PME)
- Reaction field
- Shifted/switched cutoffs (less accurate)

6.3.4 1-4 Interactions

Atoms separated by exactly 3 bonds (1-4 pairs) receive special treatment:

Force Field	**LJ 1-4 scaling**	**Coulomb 1-4 scaling**
OPLS-AA	0.5	0.5
AMBER	0.5	0.833
CHARMM	1.0 (special params)	1.0
GROMOS	1.0	1.0

6.4 Major Force Fields for Polymers

6.4.1 OPLS (Optimized Potentials for Liquid Simulations)

Developed by Jorgensen and coworkers, OPLS is optimized for liquid-state thermodynamics.

Key features:

- All-atom (OPLS-AA) and united-atom (OPLS-UA) versions
- Optimized for liquid densities and heats of vaporization
- Geometric combining rules
- 1-4 scaling: 0.5 for both LJ and Coulomb

OPLS-AA for polyethylene:

```
# OPLS-AA parameters for PE
opls_pe_params = {
    'atom_types': {
        'CT': {'mass': 12.011, 'sigma': 3.50, 'epsilon': 0.066, 'charge':
    -0.18},
        'HC': {'mass': 1.008, 'sigma': 2.50, 'epsilon': 0.030, 'charge':
    0.06}
    },
    'bonds': {
        ('CT', 'CT'): {'k': 268.0, 'r0': 1.529},
        ('CT', 'HC'): {'k': 340.0, 'r0': 1.090}
    },
    'angles': {
        ('CT', 'CT', 'CT'): {'k': 58.35, 'theta0': 112.7},
        ('CT', 'CT', 'HC'): {'k': 37.50, 'theta0': 110.7},
        ('HC', 'CT', 'HC'): {'k': 33.00, 'theta0': 107.8}
    },
    'dihedrals': {
        ('CT', 'CT', 'CT', 'CT'): {'V1': 1.740, 'V2': -0.157, 'V3': 0.279}
    }
}
```

6.4.2 AMBER (Assisted Model Building with Energy Refinement)

Originally developed for biomolecules, AMBER has been extended to synthetic polymers.

Key features:

- Strong focus on electrostatics (RESP charges)
- Lorentz-Berthelot combining rules
- GAFF (General AMBER Force Field) for organic molecules
- Well-validated for proteins and nucleic acids

GAFF for polymers:

```
# Example: GAFF parameters for PMMA
gaff_pmma_types = {
    # Backbone carbons
    'c3': {'element': 'C', 'hybridization': 'sp3'},
    'c': {'element': 'C', 'hybridization': 'sp2', 'carbonyl': True},
```

```
    # Oxygen types
    'o': {'element': 'O', 'hybridization': 'sp2', 'carbonyl': True},
    'os': {'element': 'O', 'hybridization': 'sp3', 'ether': True},
    # Hydrogen types
    'hc': {'element': 'H', 'bonded_to': 'C'}
}
```

6.4.3 CHARMM (Chemistry at HARvard Macromolecular Mechanics)

CHARMM offers extensive parameters for biological and synthetic polymers.

Key features:

- Explicit hydrogen bonding terms (in older versions)
- CMAP correction for backbone dihedrals
- CGenFF (CHARMM General Force Field) for small molecules
- Lorentz-Berthelot combining rules
- Special 1-4 LJ parameters (not just scaling)

CHARMM-style input:

```
* CHARMM topology for polyethylene
*
MASS     1 CT3    12.01100 ! methyl carbon
MASS     2 CT2    12.01100 ! methylene carbon
MASS     3 HA3     1.00800 ! methyl hydrogen
MASS     4 HA2     1.00800 ! methylene hydrogen

RESI PE     0.00  ! polyethylene repeat unit
ATOM C1   CT2  -0.18
ATOM H1   HA2   0.09
ATOM H2   HA2   0.09
BOND C1 H1  C1 H2
PATCH FIRST NONE LAST NONE
```

6.4.4 TraPPE (Transferable Potentials for Phase Equilibria)

TraPPE is specifically designed for phase equilibrium predictions.

Key features:

- United-atom model (CH_3, CH_2, CH as single sites)
- Optimized for vapor-liquid equilibria
- Excellent for density predictions
- Simple functional forms

Site	ϵ (K)	σ (Å)	**Description**
CH_4	148.0	3.73	methane
CH_3	98.0	3.75	methyl group
CH_2	46.0	3.95	methylene group
CH	10.0	4.68	methine group

6.5 Force Field Parameterization

6.5.1 Ab Initio Fitting

Parameters are fit to quantum mechanical calculations:

```
def fit_dihedral_potential(qm_energies, phi_angles, n_terms=4):
    """
    Fit Fourier dihedral coefficients to QM torsion scan.

    U(phi) = sum_n (V_n/2) * [1 + cos(n*phi - gamma_n)]
    """
    import numpy as np
    from scipy.optimize import least_squares

    def dihedral_energy(params, phi):
        V = params[:n_terms]
        gamma = params[n_terms:]
        U = np.zeros_like(phi)
        for n in range(1, n_terms + 1):
            U += (V[n-1] / 2) * (1 + np.cos(n * phi - gamma[n-1]))
        return U

    def residual(params, phi, E_qm):
        E_ff = dihedral_energy(params, phi)
        return E_ff - E_qm + np.min(E_qm) - np.min(E_ff)

    # Initial guess
    x0 = np.zeros(2 * n_terms)

    # Fit
    result = least_squares(residual, x0, args=(phi_angles, qm_energies))

    V_fitted = result.x[:n_terms]
    gamma_fitted = result.x[n_terms:]

    return V_fitted, gamma_fitted
```

6.5.2 Empirical Fitting to Experimental Data

Target properties include:

- Density
- Heat of vaporization
- Characteristic ratio C_∞
- Glass transition temperature T_g
- X-ray/neutron scattering patterns

```
def objective_function(params, experimental_data):
    """
    Multi-objective fitting to experimental properties.
    """
    # Run MD simulation with trial parameters
    traj = run_md_simulation(params)
```

```
    # Compute properties
    rho_sim = compute_density(traj)
    Rg_sim = compute_Rg(traj)
    Tg_sim = compute_Tg(traj)

    # Weighted residuals
    w_rho, w_Rg, w_Tg = 1.0, 0.5, 0.3

    residual = (
        w_rho * (rho_sim - experimental_data['rho'])**2 / experimental_data
    ['rho']**2 +
        w_Rg * (Rg_sim - experimental_data['Rg'])**2 / experimental_data['
    Rg']**2 +
        w_Tg * (Tg_sim - experimental_data['Tg'])**2 / experimental_data['
    Tg']**2
    )

    return residual
```

6.6 Practical Considerations

6.6.1 Cutoff Schemes

```
# LAMMPS cutoff settings for polymers
"""
# Standard cutoffs
pair_style lj/cut/coul/long 12.0 12.0
kspace_style pppm 1e-4

# Shifted potential (smoother energy conservation)
pair_modify shift yes

# Tail corrections for pressure/energy
pair_modify tail yes
"""
```

6.6.2 Timestep Selection

System Type	Recommended Δt	Notes
All-atom (flexible H)	0.5–1.0 fs	C–H vibrations limit
All-atom (SHAKE/LINCS)	1.0–2.0 fs	Constrain H bonds
United-atom	2.0–4.0 fs	No fast H vibrations
Coarse-grained	5.0–20.0 fs	Depends on bead mass

6.6.3 Force Field Validation

Essential validation checks:

1. **Density**: Within 2% of experimental at T, P
2. C_∞: Matches experimental characteristic ratio
3. R_g **scaling**: $R_g \sim N^{0.5}$ for theta solvent

4. T_g: Within 10–20 K of experimental (cooling rate dependent)
5. **Diffusion**: Correct temperature and MW dependence

```
def validate_force_field(traj, experimental):
    """
    Comprehensive force field validation.
    """
    results = {}

    # Density check
    rho = compute_density(traj)
    results['density'] = {
        'simulated': rho,
        'experimental': experimental['rho'],
        'error_pct': 100 * abs(rho - experimental['rho']) / experimental['
    rho']
    }

    # Characteristic ratio
    Cinf = compute_characteristic_ratio(traj)
    results['Cinf'] = {
        'simulated': Cinf,
        'experimental': experimental['Cinf'],
        'error_pct': 100 * abs(Cinf - experimental['Cinf']) / experimental[
    'Cinf']
    }

    # RDF comparison
    r, g_r = compute_rdf(traj)
    results['rdf'] = {'r': r, 'g_r': g_r}

    # Torsion distribution
    phi, P_phi = compute_torsion_distribution(traj)
    results['torsions'] = {'phi': phi, 'P_phi': P_phi}

    return results
```

6.7 Example: Setting Up a Polyethylene Simulation

```
# Complete LAMMPS input for PE melt simulation
lammps_input = """
# Initialization
units real
atom_style full
boundary p p p

# Force field
pair_style lj/cut/coul/long 12.0 12.0
bond_style harmonic
angle_style harmonic
dihedral_style opls
kspace_style pppm 1e-4

# Read structure
read_data pe_melt.data
```

```
# OPLS-AA parameters
pair_coeff 1 1 0.066 3.50   # CT-CT
pair_coeff 2 2 0.030 2.50   # HC-HC

bond_coeff 1 268.0 1.529    # CT-CT
bond_coeff 2 340.0 1.090    # CT-HC

angle_coeff 1 58.35 112.7   # CT-CT-CT
angle_coeff 2 37.50 110.7   # CT-CT-HC
angle_coeff 3 33.00 107.8   # HC-CT-HC

dihedral_coeff 1 1.740 -0.157 0.279 0.0  # CT-CT-CT-CT
dihedral_coeff 2 0.0 0.0 0.366 0.0       # CT-CT-CT-HC
dihedral_coeff 3 0.0 0.0 0.318 0.0       # HC-CT-CT-HC

# Neighbor list
neighbor 2.0 bin
neigh_modify delay 0 every 1 check yes

# Equilibration
fix 1 all nvt temp 450 450 100.0
timestep 1.0
thermo 1000
run 100000

# Production
reset_timestep 0
dump 1 all custom 1000 traj.lammpstrj id type x y z
run 1000000
"""
```

6.8 Force Field Comparison

Property	OPLS-AA	GAFF	TraPPE-UA	Experiment
PE density (g/cm^3)	0.76	0.77	0.78	0.78
PE C_∞	7.8	8.1	7.4	7.4
PS T_g (K)	380	385	–	373
Comp. cost	High	High	Low	–

6.9 Summary

Key considerations for polymer force field selection:

- Match the force field to your target properties
- All-atom for local structure, UA for large-scale conformations
- Validate against experimental benchmarks before production
- Be consistent with combining rules and 1-4 interactions
- Consider computational cost vs. accuracy tradeoffs

6.10 Exercises

1. Parameterize a dihedral potential for polypropylene by fitting to QM torsion scan data.
2. Compare density predictions from OPLS-AA and TraPPE-UA for a polyethylene melt at 450 K.
3. Derive the relationship between OPLS dihedral coefficients and Ryckaert-Bellemans coefficients.
4. Set up a validation protocol to test a new PMMA force field against experimental C_∞ and density.
5. Investigate the effect of LJ cutoff distance on polymer melt properties.

6.11 Further Reading

- Jorgensen, W.L. et al. "OPLS Force Field for Proteins"
- Wang, J. et al. "Development of GAFF"
- MacKerell, A.D. et al. "CHARMM Force Field Development"
- Martin, M.G. & Siepmann, J.I. "TraPPE Force Field"

Chapter 7

Coarse-Grained Models

7.1 Philosophy of Coarse-Graining

Coarse-graining (CG) is the systematic reduction of degrees of freedom while preserving essential physics. For polymers, this means grouping multiple atoms into single interaction sites ("beads").

Definition 7.1 (Coarse-Graining)**.** *A mapping $\mathcal{M} : \mathbb{R}^{3n} \to \mathbb{R}^{3N}$ from n atomistic coordinates to $N < n$ CG coordinates, designed to preserve target properties.*

7.1.1 Why Coarse-Grain?

Metric	Atomistic	CG (4:1)
Atoms per 100 kg/mol PE	∼21,000	∼2,500
Timestep	1–2 fs	10–50 fs
Accessible time	ns	μs
Computational speedup	1×	100–1000×

7.1.2 Trade-offs

- **Gained**: Longer times, larger systems, ensemble averaging
- **Lost**: Chemical detail, fast dynamics, specific hydrogen bonds
- **Transferred**: Chain connectivity, excluded volume, key conformational preferences

7.2 The Kremer-Grest Model

The Kremer-Grest (KG) model is the most widely used CG polymer model, designed for studying *generic* polymer behavior.

7.2.1 Model Definition

Beads interact via two potentials:

1. **Non-bonded**: WCA (purely repulsive Lennard-Jones)
2. **Bonded**: FENE (finitely extensible nonlinear elastic)

7.2.2 WCA (Weeks-Chandler-Andersen) Potential

$$\boxed{U_{\text{WCA}}(r) = \begin{cases} 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] + \varepsilon & r < 2^{1/6}\sigma \\ 0 & r \geq 2^{1/6}\sigma \end{cases}} \tag{7.1}$$

Key features:

- Purely repulsive (LJ truncated at minimum)
- Cutoff at $r_c = 2^{1/6}\sigma \approx 1.122\sigma$
- Provides excluded volume without attraction
- Smooth, continuous force at cutoff

```
def wca_potential(r, sigma, epsilon):
    """
    WCA potential (repulsive LJ).

    Args:
        r: Distance (tensor)
        sigma: Bead diameter
        epsilon: Energy scale

    Returns:
        Potential energy
    """
    r_cut = 2**(1/6) * sigma

    # Only compute for r < r_cut
    mask = r < r_cut
    sig_over_r = sigma / r
    sig6 = sig_over_r ** 6
    sig12 = sig6 ** 2

    U = torch.where(
        mask,
        4 * epsilon * (sig12 - sig6) + epsilon,
        torch.zeros_like(r)
    )
    return U
```

7.2.3 FENE (Finitely Extensible Nonlinear Elastic) Bond

$$\boxed{U_{\text{FENE}}(r) = -\frac{1}{2}kR_0^2 \ln\left(1 - \frac{r^2}{R_0^2}\right)} \tag{7.2}$$

Standard parameters: $k = 30\varepsilon/\sigma^2$, $R_0 = 1.5\sigma$
Key features:

- Diverges at $r = R_0$ (maximum extension)
- Prevents chain crossing (topological constraints)
- Combined with WCA gives average bond length $\langle r \rangle \approx 0.96\sigma$

```
def fene_potential(r, k, R0):
    """
    FENE bond potential.

    Args:
        r: Bond length
        k: Spring constant (typically 30*epsilon/sigma^2)
        R0: Maximum extension (typically 1.5*sigma)

    Returns:
        Bond energy (diverges as r -> R0)
    """
    r_over_R0_sq = (r / R0) ** 2

    # Avoid log of negative number
    arg = torch.clamp(1.0 - r_over_R0_sq, min=1e-6)

    U = -0.5 * k * R0**2 * torch.log(arg)
    return U
```

7.2.4 FENE Force

$$F_{\mathrm{FENE}}(r) = -\frac{\mathrm{d}U}{\mathrm{d}r} = \frac{kr}{1-(r/R_0)^2} \tag{7.3}$$

This is an *attractive* force toward $r = 0$, balanced by WCA repulsion.

7.3 Adding Chain Stiffness

The basic KG model produces very flexible chains ($C_\infty \approx 1.7$). Real polymers require additional stiffness.

7.3.1 Cosine Bending Potential

$$\boxed{U_{\mathrm{bend}}(\theta) = \kappa\varepsilon(1+\cos\theta)} \tag{7.4}$$

where θ is the angle formed by three consecutive beads (supplement of the backbone angle).
Physical interpretation:

- Minimum at $\theta = 180$ (straight chain)
- Maximum at $\theta = 0$ (fully bent)
- κ controls stiffness in units of ε

7.3.2 Mapping κ to C_∞

The key result from Everaers et al. (2020) relates bending stiffness to Kuhn length:

$$\boxed{\frac{l_{\mathrm{K}}(\kappa)}{\sigma} = 1.795 + 0.358\kappa + 0.172\kappa^2 + 0.019\kappa^3} \tag{7.5}$$

The characteristic ratio follows from:

$$C_\infty = \frac{l_{\mathrm{K}}}{b_{\mathrm{CC}}} \cdot \frac{b_{\mathrm{CC}}}{b_{\mathrm{CG}}} \cdot (\text{mapping factor}) \tag{7.6}$$

7.3.3 κ Values for Common Polymers

Polymer	C_∞ (exp.)	κ (KG)	l_K/σ
Polyethylene (PE)	7.0	2.16	3.7
Polypropylene (PP)	5.9	1.5	2.9
Polystyrene (PS)	10.0	0.94	2.5
PDMS	6.3	0	1.8

```python
def compute_kuhn_length(kappa: float, sigma: float) -> float:
    """
    Compute Kuhn length from bending stiffness.

    Uses Everaers et al. 2020 polynomial fit.

    Args:
        kappa: Bending stiffness parameter
        sigma: Bead diameter

    Returns:
        Kuhn length in same units as sigma
    """
    lK_over_sigma = (1.795 + 0.358 * kappa +
                     0.172 * kappa**2 + 0.019 * kappa**3)
    return lK_over_sigma * sigma


def estimate_kappa_for_Cinf(target_Cinf: float, sigma: float = 4.5,
                            b_cc: float = 1.54) -> float:
    """
    Estimate kappa needed to achieve target C_infinity.

    Args:
        target_Cinf: Target characteristic ratio
        sigma: CG bead diameter
        b_cc: C-C bond length

    Returns:
        Estimated kappa value
    """
    import numpy as np

    # Target Kuhn length
    target_lK = target_Cinf * b_cc

    # Solve: lK/sigma = 1.795 + 0.358*k + 0.172*k^2 + 0.019*k^3
    target = target_lK / sigma
    coeffs = [0.019, 0.172, 0.358, 1.795 - target]

    roots = np.roots(coeffs)
    for root in roots:
        if np.isreal(root) and 0 < root.real < 10:
            return float(root.real)

    return 2.0  # Default fallback
```

7.4 Dihedral (Torsional) Potentials

For accurate C_∞, torsional preferences are often essential.

7.4.1 Standard Dihedral Form

$$U_{\text{dihedral}}(\phi) = \sum_{n=1}^{4} K_n(1 + \cos(n\phi - \gamma_n)) \tag{7.7}$$

For polyethylene, the key terms are:

- K_1: Trans preference ($\phi = 180$ is minimum)
- K_3: Three-fold barrier (gauche minima at ± 60)

Simplified PE dihedral:

$$U(\phi) = K_1(1 + \cos\phi) + K_3(1 + \cos 3\phi) \tag{7.8}$$

7.4.2 Physical Interpretation

Angle ϕ	Conformation	Energy
0	cis (eclipsed)	Maximum
± 60	gauche$\pm$	Local minimum
180	trans	Global minimum

For PE, gauche is $\sim$0.5–1.0 kcal/mol above trans. This preference contributes significantly to C_∞.

```
def dihedral_potential(phi, K1, K3):
    """
    Compute dihedral potential U = K1(1+cos(phi)) + K3(1+cos(3*phi)).

    Args:
        phi: Dihedral angle (radians)
        K1: Trans preference coefficient
        K3: Three-fold barrier coefficient

    Returns:
        Dihedral energy
    """
    U = K1 * (1 + torch.cos(phi)) + K3 * (1 + torch.cos(3 * phi))
    return U
```

7.5 Systematic Coarse-Graining Methods

7.5.1 Iterative Boltzmann Inversion (IBI)

IBI iteratively refines CG potentials to match atomistic distributions:

$$U^{(n+1)}(r) = U^{(n)}(r) + k_B T \ln \frac{P_{\text{CG}}^{(n)}(r)}{P_{\text{target}}(r)} \tag{7.9}$$

Algorithm:

1. Run atomistic simulation, compute target distribution $P_{\text{target}}(r)$
2. Initial guess: $U^{(0)}(r) = -k_{\text{B}}T \ln P_{\text{target}}(r)$
3. Run CG simulation with $U^{(n)}$, compute $P_{\text{CG}}^{(n)}(r)$
4. Update potential using equation above
5. Repeat until convergence

7.5.2 Force Matching

Match CG forces to atomistic forces averaged over mapped configurations:

$$\chi^2 = \sum_i \left| \mathbf{F}_i^{\text{CG}} - \langle \mathbf{F}_i^{\text{atom}} \rangle_{\mathcal{M}} \right|^2 \tag{7.10}$$

Minimize χ^2 to find optimal CG force field parameters.

7.5.3 Relative Entropy Minimization

Minimize the information loss:

$$S_{\text{rel}} = \int P_{\text{atom}}(\mathbf{r}) \ln \frac{P_{\text{atom}}(\mathbf{r})}{P_{\text{CG}}(\mathcal{M}(\mathbf{r}))} \, d\mathbf{r} \tag{7.11}$$

7.6 Multi-Scale Mapping

7.6.1 Defining the CG Mapping

For PE with 4 CH_2 units per bead:

```python
def create_cg_mapping(n_monomers: int, beads_per_monomer: int = 4):
    """
    Create coarse-graining mapping matrix.

    For PE: 4 CH2 -> 1 bead (center of mass)

    Args:
        n_monomers: Number of CH2 units
        beads_per_monomer: Mapping ratio

    Returns:
        n_beads: Number of CG beads
        mapping: (n_beads, n_atoms) mapping matrix
    """
    atoms_per_monomer = 3  # C + 2H
    n_atoms = n_monomers * atoms_per_monomer
    n_beads = n_monomers // beads_per_monomer

    # Mass-weighted center of mass mapping
    mapping = np.zeros((n_beads, n_atoms))

    for i in range(n_beads):
        start = i * beads_per_monomer * atoms_per_monomer
        end = start + beads_per_monomer * atoms_per_monomer
        # Simplified: equal weights (should be mass-weighted)
        mapping[i, start:end] = 1.0 / (end - start)
```

```
    return n_beads, mapping
```

7.6.2 Mapping Properties

When coarse-graining, intensive properties should be preserved:

- Density: $\rho_{\text{CG}} = \rho_{\text{atom}}$
- R_g: Should match (chain-level property)
- C_∞: Match via bending stiffness κ

Extensive properties scale:

- Mass: $m_{\text{bead}} = m_{\text{monomer}} \times$ mapping ratio
- Energy: Often rescaled (ε becomes effective parameter)
- Time: CG dynamics are faster (no hydrogen vibrations)

7.7 Time Mapping

CG simulations have accelerated dynamics. The time mapping factor α relates CG time to real time:

$$t_{\text{real}} = \alpha \cdot t_{\text{CG}} \tag{7.12}$$

7.7.1 Friction-Based Mapping

From Langevin dynamics:

$$\alpha = \frac{\zeta_{\text{atom}}}{\zeta_{\text{CG}}} \tag{7.13}$$

7.7.2 Diffusion-Based Mapping

Match diffusion coefficients:

$$\alpha = \frac{D_{\text{CG}}}{D_{\text{atom}}} \tag{7.14}$$

Typical values: $\alpha \sim 10$–100 for 4:1 CG mapping.

7.8 Complete KG Force Field

Putting it all together:

```
@dataclass
class KremerGrestParams:
    """Parameters for Kremer-Grest CG model."""

    # Non-bonded (WCA)
    sigma: float = 4.5      # Bead diameter (Angstrom)
    epsilon: float = 0.5    # Energy scale (kcal/mol)

    # Bonded (FENE)
    k_fene: float = 30.0    # Spring constant (epsilon/sigma^2)
    R0: float = 1.5         # Max extension (sigma units)
```

```
    # Bending stiffness
    kappa: float = 2.16     # For PE (dimensionless)

    # Dihedral (optional)
    kappa_d1: float = 1.2   # Trans preference (kcal/mol)
    kappa_d3: float = 0.4   # Three-fold barrier (kcal/mol)

    @property
    def kuhn_length(self) -> float:
        """Compute theoretical Kuhn length."""
        lK_over_sigma = (1.795 + 0.358 * self.kappa +
                         0.172 * self.kappa**2 +
                         0.019 * self.kappa**3)
        return lK_over_sigma * self.sigma

    @property
    def theoretical_Cinf(self) -> float:
        """Estimate C_infinity from kappa."""
        b_cc = 1.54  # C-C bond length
        return self.kuhn_length / b_cc
```

7.9 Validation Protocol

A CG model should be validated against:

1. **Structural properties**:
   - R_g vs chain length
   - C_∞ from R_{ee}
   - Bond/angle/dihedral distributions
2. **Thermodynamic properties**:
   - Density at target T, P
   - Equation of state
3. **Dynamic properties** (with time mapping):
   - Diffusion coefficient
   - Relaxation times

```
def validate_cg_model(md_engine, backbone_length: int,
                      target_Cinf: float = 7.0) -> dict:
    """
    Comprehensive validation of CG model.

    Args:
        md_engine: MD simulation object
        backbone_length: Number of backbone beads
        target_Cinf: Experimental C_infinity to match

    Returns:
        Validation results dictionary
    """
    # Measure from simulation
```

```
    Rg = md_engine.compute_radius_of_gyration()
    Ree = md_engine.compute_end_to_end_distance(backbone_length)
    corr = md_engine.compute_bond_vector_correlation()

    # Compute C_n
    n_bonds = backbone_length - 1
    b_cg = md_engine.params.bond_r0
    Cn_measured = Ree**2 / (n_bonds * b_cg**2)

    # Theoretical prediction
    lK_theory = md_engine.compute_kuhn_length_theoretical()
    Cn_theory = lK_theory / 1.54  # vs C-C bond

    # Gaussian chain check
    Ree2_Rg2_ratio = (Ree / Rg)**2  # Should be ~6

    return {
        'Rg': Rg,
        'Ree': Ree,
        'Cn_measured': Cn_measured,
        'Cn_theoretical': Cn_theory,
        'Cn_target': target_Cinf,
        'Ree2_Rg2_ratio': Ree2_Rg2_ratio,
        'gaussian_check': abs(Ree2_Rg2_ratio - 6) < 1,
        'Cn_error': abs(Cn_measured - target_Cinf) / target_Cinf,
        'persistence_length': corr['persistence_length_fitted'],
    }
```

7.10 Common Issues and Solutions

Issue	Symptom	Solution
C_∞ too low	Chain too compact	Increase κ, add dihedrals
C_∞ too high	Chain too extended	Decrease κ
Chain crossing	Unphysical topology	Use FENE + WCA, reduce timestep
Bond breaking	FENE divergence	Reduce timestep, minimize first
Poor equilibration	R_g drift	Longer equilibration, push-off

7.11 Summary

Component	Role
WCA potential	Excluded volume (beads can't overlap)
FENE bonds	Chain connectivity (can't break/cross)
Cosine bending	Chain stiffness (controls C_∞)
Dihedral potential	Torsional preferences (trans/gauche)
κ parameter	Key control for characteristic ratio

7.12 Exercises

1. Using the Everaers polynomial, calculate the κ value needed for polystyrene ($C_\infty = 10$).

2. Derive the force from the FENE potential and show that it is attractive.

3. Write a function to compute the dihedral angle ϕ given four consecutive bead positions.

4. Explain why the FENE + WCA combination prevents chain crossing. What would happen with harmonic bonds?

5. Design a validation test to check if a CG model correctly reproduces C_∞ for PE.

7.13 Key References

- Kremer, K. & Grest, G.S. (1990). "Dynamics of entangled linear polymer melts," J. Chem. Phys. **92**, 5057.

- Everaers, R. et al. (2020). "Kremer-Grest Models for Commodity Polymer Melts," Macromolecules **53**, 1901.

- Svaneborg, C. et al. (2020). "Characteristic Time and Length Scales in Melts of Kremer-Grest Bead-Spring Polymers," Macromolecules **53**, 1917.

- Noid, W.G. (2013). "Perspective: Coarse-grained models for biomolecular systems," J. Chem. Phys. **139**, 090901.

Chapter 8

Enhanced Sampling and Equilibration

8.1 The Equilibration Challenge in Polymer Simulations

Equilibrating polymer systems is one of the most challenging aspects of MD simulation. Long chains have astronomically long relaxation times, making brute-force equilibration impractical for high molecular weight systems.

Definition 8.1 (Equilibration Time)**.** *The equilibration time is the time required for a system to lose memory of its initial configuration and sample the equilibrium distribution. For polymers, this scales as* $\tau_{eq} \sim N^{3.4}$ *for entangled systems.*

8.1.1 Time Scale Problem

For a typical entangled polymer melt:

Polymer	N	τ_{eq} (s)	**MD accessible?**
PE, $M = 1$ kg/mol	36	10^{-9}	Yes
PE, $M = 10$ kg/mol	360	10^{-6}	Marginal
PE, $M = 100$ kg/mol	3600	10^{-3}	No
PE, $M = 1$ Mg/mol	36000	10^{0}	No

8.2 Initial Configuration Generation

8.2.1 Random Walk Generation

The simplest approach generates chains as random walks:

```
def generate_random_walk_chain(n_beads, bond_length=1.54, box_size=100.0):
    """
    Generate a single polymer chain as a random walk.
    """
    import numpy as np

    positions = np.zeros((n_beads, 3))
    positions[0] = np.random.rand(3) * box_size

    for i in range(1, n_beads):
        # Random direction on unit sphere
        theta = np.arccos(2 * np.random.rand() - 1)
        phi = 2 * np.pi * np.random.rand()

        direction = np.array([
```

```
            np.sin(theta) * np.cos(phi),
            np.sin(theta) * np.sin(phi),
            np.cos(theta)
        ])

        positions[i] = positions[i-1] + bond_length * direction

    # Apply periodic boundary conditions
    positions = positions % box_size

    return positions
```

Problem: Random walk configurations have severe overlaps at melt density.

8.2.2 Self-Avoiding Random Walk

```
def generate_saw_chain(n_beads, bond_length, min_distance, max_attempts
    =1000):
    """
    Generate self-avoiding random walk chain.
    """
    import numpy as np
    from scipy.spatial import KDTree

    positions = [np.zeros(3)]

    for i in range(1, n_beads):
        for attempt in range(max_attempts):
            # Random direction
            theta = np.arccos(2 * np.random.rand() - 1)
            phi = 2 * np.pi * np.random.rand()

            direction = np.array([
                np.sin(theta) * np.cos(phi),
                np.sin(theta) * np.sin(phi),
                np.cos(theta)
            ])

            new_pos = positions[-1] + bond_length * direction

            # Check for overlaps
            if len(positions) > 1:
                tree = KDTree(positions)
                distances, _ = tree.query(new_pos, k=min(3, len(positions))
    )
                if np.min(distances) < min_distance:
                    continue

            positions.append(new_pos)
            break
        else:
            raise RuntimeError(f"Failed to place bead {i}")

    return np.array(positions)
```

8.2.3 Semi-Crystalline Initialization

For crystallizable polymers, start from extended chains:

```
def generate_extended_chains(n_chains, n_beads, bond_length,
                             angle=112.7, box_dims):
    """
    Generate extended (all-trans) polymer chains in a layered structure.
    """
    import numpy as np

    all_positions = []

    # Chain spacing
    chain_spacing = 4.5  # Angstrom
    chains_per_row = int(np.sqrt(n_chains))

    for chain_idx in range(n_chains):
        ix = chain_idx % chains_per_row
        iy = chain_idx // chains_per_row

        positions = np.zeros((n_beads, 3))

        # Starting position
        positions[0] = [ix * chain_spacing, iy * chain_spacing, 0]

        # Build extended chain along z
        theta_rad = np.radians(angle)
        dz = bond_length * np.cos((np.pi - theta_rad) / 2)
        dx = bond_length * np.sin((np.pi - theta_rad) / 2)

        for i in range(1, n_beads):
            positions[i] = positions[i-1].copy()
            positions[i, 2] += dz
            positions[i, 0] += dx * ((-1) ** i)

        all_positions.append(positions)

    return np.array(all_positions)
```

8.3 Push-Off Methods

8.3.1 Soft-Core Potential Push-Off

Replace the hard LJ potential with a soft repulsive potential during initial equilibration:

$$\boxed{U_{\text{soft}}(r) = A\left[1 + \cos\left(\frac{\pi r}{r_c}\right)\right] \quad \text{for } r < r_c} \tag{8.1}$$

```
# LAMMPS soft potential push-off
"""
# Soft potential for overlap removal
pair_style soft 1.5
pair_coeff * * 100.0 1.5

# Gradually increase repulsion
variable prefactor equal ramp(0,100)
```

```
fix soft all adapt 1 pair soft a * * v_prefactor

# Energy minimization
minimize 1e-4 1e-6 1000 10000

# NVT with soft potential
fix nvt all nvt temp 500 500 100
run 10000

# Switch to full LJ potential
pair_style lj/cut 12.0
pair_coeff * * 0.066 3.50
run 50000
"""
```

8.3.2 Gradual Potential Ramping

```
def push_off_protocol(system, n_stages=10, initial_epsilon=0.001):
    """
    Gradual push-off by ramping LJ epsilon.
    """
    import numpy as np

    epsilon_schedule = np.logspace(np.log10(initial_epsilon), 0, n_stages)

    for stage, eps_factor in enumerate(epsilon_schedule):
        # Scale all epsilon values
        for pair in system.pair_interactions:
            pair.epsilon = pair.epsilon_full * eps_factor

        # Short equilibration at each stage
        system.run(steps=10000, ensemble='nvt', temperature=500)

        print(f"Stage {stage+1}/{n_stages}: eps_factor = {eps_factor:.4f}")
        print(f"  Max force: {system.max_force:.2f}")
        print(f"  Potential energy: {system.potential_energy:.2f}")
```

8.4 Double-Bridging Algorithm

The double-bridging algorithm efficiently equilibrates long chains by performing nonphysical chain reconnections that preserve topology but accelerate relaxation.

8.4.1 Algorithm Description

1. Select two pairs of bonded monomers: $(i, i+1)$ and $(j, j+1)$ on different chains
2. Check geometric criteria for swap feasibility
3. Perform double bridge: connect i to $j+1$ and j to $i+1$
4. Accept/reject based on energy change (Metropolis criterion)

```
def double_bridging_move(chains, bond_list, cutoff=2.0, kT=1.0):
    """
    Attempt a double-bridging Monte Carlo move.
```

```
    Swaps bonded pairs between chains to accelerate equilibration.
    """
    import numpy as np

    n_chains = len(chains)

    # Select two different chains
    chain_a, chain_b = np.random.choice(n_chains, 2, replace=False)

    # Select bonds on each chain (not end bonds)
    n_a = len(chains[chain_a]) - 2
    n_b = len(chains[chain_b]) - 2

    if n_a < 1 or n_b < 1:
        return False

    bond_a = np.random.randint(1, n_a)  # monomer i
    bond_b = np.random.randint(1, n_b)  # monomer j

    # Get positions
    pos_i = chains[chain_a][bond_a]
    pos_i1 = chains[chain_a][bond_a + 1]
    pos_j = chains[chain_b][bond_b]
    pos_j1 = chains[chain_b][bond_b + 1]

    # Check if new bonds are geometrically feasible
    dist_i_j1 = np.linalg.norm(pos_i - pos_j1)
    dist_j_i1 = np.linalg.norm(pos_j - pos_i1)

    if dist_i_j1 > cutoff or dist_j_i1 > cutoff:
        return False

    # Calculate energy change (simplified)
    old_bond_energy = bond_energy(pos_i, pos_i1) + bond_energy(pos_j,
   pos_j1)
    new_bond_energy = bond_energy(pos_i, pos_j1) + bond_energy(pos_j,
   pos_i1)

    delta_E = new_bond_energy - old_bond_energy

    # Metropolis criterion
    if delta_E < 0 or np.random.rand() < np.exp(-delta_E / kT):
        # Perform swap
        perform_chain_swap(chains, chain_a, bond_a, chain_b, bond_b)
        return True

    return False
```

8.4.2 Implementation in LAMMPS

The double-bridging algorithm can be implemented using LAMMPS' fix bond/swap:

```
# LAMMPS double-bridging equilibration
"""
# Define bond swap fix
fix dbswap all bond/swap 100 0.5 1.5 1234567 &
    semi_angle 45 seed 12345
```

```
# Run with swap moves
run 1000000

# Disable swaps for production
unfix dbswap
"""
```

8.5 Chain Connectivity Altering Monte Carlo

8.5.1 End-Bridging Algorithm

End-bridging moves transfer chain ends between nearby monomers:

```
def end_bridging_move(chains, kT=1.0, cutoff=3.0):
    """
    End-bridging MC move: transfer chain end to nearby monomer.
    """
    import numpy as np

    n_chains = len(chains)

    # Select a chain and its end
    donor_chain = np.random.randint(n_chains)
    end = np.random.choice(['head', 'tail'])

    if end == 'head':
        donor_pos = chains[donor_chain][0]
    else:
        donor_pos = chains[donor_chain][-1]

    # Find nearby monomers on other chains
    candidates = []
    for c_idx, chain in enumerate(chains):
        if c_idx == donor_chain:
            continue
        for m_idx, pos in enumerate(chain[1:-1], start=1):  # exclude ends
            dist = np.linalg.norm(pos - donor_pos)
            if dist < cutoff:
                candidates.append((c_idx, m_idx, dist))

    if not candidates:
        return False

    # Select target randomly
    target_chain, target_mono, _ = candidates[np.random.randint(len(
   candidates))]

    # Calculate energy change and apply Metropolis
    # ... (similar to double-bridging)

    return True
```

8.6 Hierarchical Equilibration Strategies

8.6.1 Multi-Scale Approach

1. **Coarse-grained equilibration**: Equilibrate a CG model (faster)
2. **Backmapping**: Convert CG configuration to all-atom
3. **Local relaxation**: Short all-atom MD to relax local structure

```
def hierarchical_equilibration(target_system):
    """
    Multi-scale equilibration protocol.
    """
    # Step 1: Create CG representation
    cg_system = create_cg_system(target_system.topology)

    # Step 2: Fast CG equilibration
    cg_system.run(steps=1000000, dt=20.0)  # Large timestep

    # Step 3: Backmap to atomistic
    aa_positions = backmap_cg_to_aa(cg_system.positions,
                                    target_system.topology)

    # Step 4: Local relaxation
    target_system.positions = aa_positions
    target_system.minimize()
    target_system.run(steps=100000, dt=1.0)

    return target_system
```

8.6.2 Progressive Chain Growth

Grow chains gradually rather than starting with full-length chains:

```
def progressive_chain_growth(n_chains, final_length, growth_rate=10):
    """
    Equilibrate by growing chains progressively.
    """
    current_length = 10  # Start with short chains

    while current_length < final_length:
        # Create system with current chain length
        system = create_polymer_system(n_chains, current_length)

        # Equilibrate
        system.run_equilibration(steps=current_length * 1000)

        # Grow chains by adding monomers
        current_length = min(current_length + growth_rate, final_length)
        system = extend_chains(system, current_length)

        print(f"Chain length: {current_length}/{final_length}")

    # Final equilibration
    system.run_equilibration(steps=final_length * 10000)

    return system
```

8.7 Equilibration Verification

8.7.1 End-to-End Vector Decorrelation

```
def check_ree_decorrelation(trajectory, n_chains):
    """
    Check if end-to-end vectors have decorrelated.
    """
    import numpy as np

    Ree_vectors = []

    for frame in trajectory:
        frame_Ree = []
        for chain_idx in range(n_chains):
            chain_pos = frame.positions[chain_idx]
            Ree = chain_pos[-1] - chain_pos[0]
            frame_Ree.append(Ree)
        Ree_vectors.append(frame_Ree)

    Ree_vectors = np.array(Ree_vectors)

    # Compute autocorrelation
    n_frames = len(Ree_vectors)
    C_ree = np.zeros(n_frames // 2)

    for chain in range(n_chains):
        Ree_chain = Ree_vectors[:, chain, :]
        Ree_dot = np.sum(Ree_chain * Ree_chain, axis=1)

        for tau in range(len(C_ree)):
            C_ree[tau] += np.mean(
                np.sum(Ree_chain[:n_frames-tau] * Ree_chain[tau:], axis=1)
            ) / np.mean(Ree_dot)

    C_ree /= n_chains

    # Check if decorrelated (C < 0.1)
    is_equilibrated = C_ree[-1] < 0.1

    return C_ree, is_equilibrated
```

8.7.2 Multiple Metric Verification

```
def comprehensive_equilibration_check(trajectory):
    """
    Check multiple metrics for equilibration.
    """
    results = {}

    # 1. Energy stability
    energies = [frame.potential_energy for frame in trajectory]
    results['energy_stable'] = is_stable(energies)

    # 2. Density stability
    densities = [frame.density for frame in trajectory]
```

```
    results['density_stable'] = is_stable(densities)

    # 3. Rg convergence
    Rg_values = [compute_Rg(frame) for frame in trajectory]
    results['Rg_converged'] = is_stable(Rg_values)

    # 4. End-to-end decorrelation
    C_ree, ree_decorr = check_ree_decorrelation(trajectory, n_chains)
    results['Ree_decorrelated'] = ree_decorr

    # 5. MSD shows diffusive behavior
    msd = compute_msd(trajectory)
    results['diffusive'] = check_diffusive_msd(msd)

    # Overall assessment
    results['equilibrated'] = all([
        results['energy_stable'],
        results['density_stable'],
        results['Rg_converged'],
        results['Ree_decorrelated']
    ])

    return results


def is_stable(values, window=0.2):
    """Check if values are stable (no drift)."""
    import numpy as np
    n = len(values)
    first_half = np.mean(values[:n//2])
    second_half = np.mean(values[n//2:])
    return abs(first_half - second_half) / first_half < window
```

8.8 Practical Equilibration Protocol

```
def equilibration_protocol(system, target_density, target_temp):
    """
    Complete equilibration protocol for polymer melts.
    """
    # Stage 1: Energy minimization
    print("Stage 1: Energy minimization")
    system.minimize(max_steps=10000, force_tolerance=1e-4)

    # Stage 2: Soft push-off
    print("Stage 2: Soft potential push-off")
    system.set_soft_potential()
    system.run_nvt(steps=50000, temp=target_temp)

    # Stage 3: Switch to full potential
    print("Stage 3: Full potential equilibration")
    system.set_full_potential()
    system.run_nvt(steps=100000, temp=target_temp * 1.5)  # High T first

    # Stage 4: NPT to target density
    print("Stage 4: NPT equilibration")
    system.run_npt(steps=500000, temp=target_temp, press=1.0)
```

```
    # Stage 5: NVT at target density
    print("Stage 5: Final NVT equilibration")
    system.run_nvt(steps=1000000, temp=target_temp)

    # Check equilibration
    print("Checking equilibration...")
    check_results = comprehensive_equilibration_check(system.trajectory)

    if not check_results['equilibrated']:
        print("WARNING: System may not be fully equilibrated")
        print(f"Results: {check_results}")

    return system
```

8.9 Summary

Method	Best Use Case
Soft push-off	Initial overlap removal, any system
Double-bridging	Long entangled chains, melt equilibration
End-bridging	Polydisperse systems, chain length exchange
Hierarchical	Very long chains, atomistic detail needed
Progressive growth	Networks, crosslinked systems

8.10 Exercises

1. Implement a soft-potential push-off and compare overlap removal efficiency with direct LJ equilibration.
2. Design a double-bridging protocol for a polystyrene melt. What acceptance rates do you expect?
3. Compare equilibration times for a 500-bead chain using brute-force MD vs. double-bridging.
4. Develop criteria to determine when a polymer system is sufficiently equilibrated.
5. Implement a hierarchical equilibration scheme using a simple CG model.

8.11 Further Reading

- Auhl, R. et al. “Equilibration of long chain polymer melts in computer simulations”
- Karayiannis, N.C. et al. “Monte Carlo scheme for generation of polymer melts”
- Kremer, K. & Grest, G.S. “Dynamics of entangled linear polymer melts”

Part III

Analysis and Property Prediction

Chapter 9

Structural Analysis

9.1 Introduction to Structural Characterization

Structural analysis of polymer simulations allows direct comparison with experimental scattering data and provides insights into chain conformations, local packing, and morphology. This chapter covers the key structural metrics and their computation from MD trajectories.

Definition 9.1 (Structural Analysis)**.** *Structural analysis encompasses the quantitative characterization of molecular organization, including chain dimensions, orientations, local density, and spatial correlations.*

9.2 Chain Dimensions

9.2.1 End-to-End Distance

The end-to-end distance connects the first and last monomers:

$$\boxed{R_{\text{ee}} = |\mathbf{R}_N - \mathbf{R}_1|} \tag{9.1}$$

The mean-square end-to-end distance:

$$\langle R_{ee}^2 \rangle = \langle (\mathbf{R}_N - \mathbf{R}_1)^2 \rangle \tag{9.2}$$

For an ideal chain:

$$\langle R_{ee}^2 \rangle = N l^2 C_\infty = N_K l_K^2 \tag{9.3}$$

```python
def compute_end_to_end_distance(positions):
    """
    Compute end-to-end distance for polymer chains.

    Args:
        positions: Array of shape (n_chains, n_beads, 3)

    Returns:
        Ree for each chain, Ree^2 average
    """
    import numpy as np

    # Vector from first to last bead
    Ree_vectors = positions[:, -1, :] - positions[:, 0, :]

    # Handle periodic boundaries if needed
```

```
    # Ree_vectors = minimum_image(Ree_vectors, box)

    # Magnitudes
    Ree = np.linalg.norm(Ree_vectors, axis=1)

    Ree_squared_avg = np.mean(Ree**2)

    return Ree, Ree_squared_avg
```

9.2.2 Radius of Gyration

The radius of gyration measures the overall chain size:

$$\boxed{R_g^2 = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{R}_i - \mathbf{R}_{cm})^2} \tag{9.4}$$

where $\mathbf{R}_{cm} = \frac{1}{N}\sum_i \mathbf{R}_i$ is the center of mass.
Alternative formulation (avoids COM calculation):

$$R_g^2 = \frac{1}{2N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\mathbf{R}_i - \mathbf{R}_j)^2 \tag{9.5}$$

For ideal chains:

$$\langle R_g^2 \rangle = \frac{\langle R_{ee}^2 \rangle}{6} = \frac{Nl^2 C_\infty}{6} \tag{9.6}$$

```
def compute_radius_of_gyration(positions, masses=None):
    """
    Compute radius of gyration for polymer chains.

    Args:
        positions: Array of shape (n_chains, n_beads, 3)
        masses: Optional mass array of shape (n_chains, n_beads)

    Returns:
        Rg for each chain, average Rg^2
    """
    import numpy as np

    n_chains, n_beads, _ = positions.shape

    if masses is None:
        masses = np.ones((n_chains, n_beads))

    Rg_squared = []

    for chain_idx in range(n_chains):
        pos = positions[chain_idx]
        mass = masses[chain_idx]
        total_mass = np.sum(mass)

        # Center of mass
        com = np.sum(pos * mass[:, np.newaxis], axis=0) / total_mass

        # Squared distances from COM
        r_sq = np.sum((pos - com)**2, axis=1)
```

```
        # Mass-weighted Rg^2
        Rg2 = np.sum(mass * r_sq) / total_mass
        Rg_squared.append(Rg2)

    Rg_squared = np.array(Rg_squared)
    Rg = np.sqrt(Rg_squared)

    return Rg, np.mean(Rg_squared)
```

9.2.3 Characteristic Ratio

The characteristic ratio C_∞ quantifies chain stiffness:

$$\boxed{C_\infty = \lim_{N\to\infty} \frac{\langle R_{ee}^2 \rangle}{Nl^2}} \tag{9.7}$$

where l is the backbone bond length.

```
def compute_characteristic_ratio(positions, bond_length=1.54):
    """
    Compute characteristic ratio Cinf from chain conformations.
    """
    import numpy as np

    n_chains, n_beads, _ = positions.shape
    N = n_beads - 1  # Number of bonds

    # Compute <Ree^2>
    Ree_vectors = positions[:, -1, :] - positions[:, 0, :]
    Ree_squared_avg = np.mean(np.sum(Ree_vectors**2, axis=1))

    # Characteristic ratio
    Cinf = Ree_squared_avg / (N * bond_length**2)

    return Cinf
```

Typical values:

Polymer	C_∞
Polyethylene	6.7–7.4
Polypropylene (isotactic)	5.5–6.0
Polystyrene (atactic)	9–10
PMMA (atactic)	7–9
Polydimethylsiloxane	6.2–6.8

9.3 Chain Shape Analysis

9.3.1 Gyration Tensor

The gyration tensor provides information about chain shape:

$$S_{\alpha\beta} = \frac{1}{N}\sum_{i=1}^{N}(R_{i\alpha} - R_{cm,\alpha})(R_{i\beta} - R_{cm,\beta}) \tag{9.8}$$

Eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3$ of $\mathbf{S}$ give principal moments with:

$$R_g^2 = \lambda_1 + \lambda_2 + \lambda_3 \tag{9.9}$$

9.3.2 Shape Parameters

Asphericity:

$$b = \lambda_1 - \frac{1}{2}(\lambda_2 + \lambda_3) \tag{9.10}$$

Acylindricity:

$$c = \lambda_2 - \lambda_3 \tag{9.11}$$

Relative shape anisotropy:

$$\kappa^2 = \frac{b^2 + (3/4)c^2}{R_g^4} \tag{9.12}$$

Values: $\kappa^2 = 0$ (sphere), $\kappa^2 = 1$ (rod), $\kappa^2 = 1/4$ (planar).

```python
def compute_shape_parameters(positions):
    """
    Compute shape parameters from gyration tensor.
    """
    import numpy as np

    n_chains, n_beads, _ = positions.shape

    results = {
        'asphericity': [],
        'acylindricity': [],
        'kappa_squared': [],
        'eigenvalues': []
    }

    for chain_idx in range(n_chains):
        pos = positions[chain_idx]
        com = np.mean(pos, axis=0)
        pos_centered = pos - com

        # Gyration tensor
        S = np.zeros((3, 3))
        for i in range(n_beads):
            S += np.outer(pos_centered[i], pos_centered[i])
        S /= n_beads

        # Eigenvalues (sorted descending)
        eigenvalues = np.sort(np.linalg.eigvalsh(S))[::-1]
        l1, l2, l3 = eigenvalues

        # Shape parameters
        Rg2 = l1 + l2 + l3
        b = l1 - 0.5 * (l2 + l3)  # Asphericity
        c = l2 - l3               # Acylindricity
        kappa2 = (b**2 + 0.75 * c**2) / Rg2**2  # Relative anisotropy

        results['asphericity'].append(b)
        results['acylindricity'].append(c)
        results['kappa_squared'].append(kappa2)
        results['eigenvalues'].append(eigenvalues)

    # Convert to arrays and compute averages
    for key in results:
        results[key] = np.array(results[key])
```

```
    results['avg_kappa_squared'] = np.mean(results['kappa_squared'])

    return results
```

9.4 Bond and Torsional Analysis

9.4.1 Bond Length Distribution

```
def compute_bond_length_distribution(positions, n_bins=100):
    """
    Compute distribution of bond lengths.
    """
    import numpy as np

    n_chains, n_beads, _ = positions.shape

    bond_lengths = []

    for chain_idx in range(n_chains):
        for i in range(n_beads - 1):
            bond_vec = positions[chain_idx, i+1] - positions[chain_idx, i]
            bond_lengths.append(np.linalg.norm(bond_vec))

    bond_lengths = np.array(bond_lengths)

    hist, bin_edges = np.histogram(bond_lengths, bins=n_bins, density=True)
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

    return bin_centers, hist, np.mean(bond_lengths), np.std(bond_lengths)
```

9.4.2 Bond Angle Distribution

```
def compute_angle_distribution(positions, n_bins=180):
    """
    Compute distribution of bond angles.
    """
    import numpy as np

    n_chains, n_beads, _ = positions.shape

    angles = []

    for chain_idx in range(n_chains):
        for i in range(n_beads - 2):
            v1 = positions[chain_idx, i] - positions[chain_idx, i+1]
            v2 = positions[chain_idx, i+2] - positions[chain_idx, i+1]

            cos_angle = np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.
    norm(v2))
            cos_angle = np.clip(cos_angle, -1, 1)
            angle = np.degrees(np.arccos(cos_angle))
            angles.append(angle)

    angles = np.array(angles)
```

```
    hist, bin_edges = np.histogram(angles, bins=n_bins, range=(0, 180),
    density=True)
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

    return bin_centers, hist
```

9.4.3 Dihedral Angle Distribution

```
def compute_dihedral_distribution(positions, n_bins=360):
    """
    Compute distribution of backbone dihedral angles.
    """
    import numpy as np

    def compute_dihedral(p1, p2, p3, p4):
        """Compute dihedral angle for four points."""
        b1 = p2 - p1
        b2 = p3 - p2
        b3 = p4 - p3

        n1 = np.cross(b1, b2)
        n2 = np.cross(b2, b3)

        n1 /= np.linalg.norm(n1)
        n2 /= np.linalg.norm(n2)

        m1 = np.cross(n1, b2 / np.linalg.norm(b2))

        x = np.dot(n1, n2)
        y = np.dot(m1, n2)

        return np.degrees(np.arctan2(y, x))

    n_chains, n_beads, _ = positions.shape

    dihedrals = []

    for chain_idx in range(n_chains):
        for i in range(n_beads - 3):
            phi = compute_dihedral(
                positions[chain_idx, i],
                positions[chain_idx, i+1],
                positions[chain_idx, i+2],
                positions[chain_idx, i+3]
            )
            dihedrals.append(phi)

    dihedrals = np.array(dihedrals)

    hist, bin_edges = np.histogram(dihedrals, bins=n_bins,
                                   range=(-180, 180), density=True)
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

    # Compute trans/gauche fractions
    trans_fraction = np.sum(np.abs(dihedrals) > 150) / len(dihedrals)
    gauche_fraction = np.sum(np.abs(np.abs(dihedrals) - 65) < 25) / len(
    dihedrals)
```

```
    return bin_centers, hist, trans_fraction, gauche_fraction
```

9.5 Radial Distribution Functions

9.5.1 Pair Distribution Function

The pair distribution function $g(r)$ gives the probability of finding a particle at distance r:

$$\boxed{g(r) = \frac{V}{N^2} \langle \sum_{i \neq j} \delta(r - |\mathbf{r}_i - \mathbf{r}_j|) \rangle} \tag{9.13}$$

```
def compute_rdf(positions, box_size, n_bins=200, r_max=None):
    """
    Compute radial distribution function g(r).
    """
    import numpy as np
    from scipy.spatial.distance import pdist

    if r_max is None:
        r_max = box_size / 2

    n_particles = len(positions)

    # Compute all pairwise distances
    distances = pdist(positions)

    # Apply minimum image convention
    distances = distances - box_size * np.round(distances / box_size)
    distances = np.abs(distances)

    # Histogram
    hist, bin_edges = np.histogram(distances, bins=n_bins, range=(0, r_max)
    )
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])
    dr = bin_edges[1] - bin_edges[0]

    # Normalize
    volume = box_size**3
    rho = n_particles / volume

    # Shell volumes
    shell_volumes = 4 * np.pi * bin_centers**2 * dr

    # Ideal gas reference
    n_ideal = rho * shell_volumes * (n_particles - 1) / 2

    g_r = hist / n_ideal

    return bin_centers, g_r
```

9.5.2 Intermolecular vs. Intramolecular RDF

```python
def compute_inter_intra_rdf(positions, chain_ids, box_size, n_bins=200):
    """
    Compute separate inter- and intra-molecular RDFs.
    """
    import numpy as np

    n_particles = len(positions)
    r_max = box_size / 2

    inter_distances = []
    intra_distances = []

    for i in range(n_particles):
        for j in range(i+1, n_particles):
            r_vec = positions[j] - positions[i]
            # Minimum image
            r_vec = r_vec - box_size * np.round(r_vec / box_size)
            r = np.linalg.norm(r_vec)

            if r < r_max:
                if chain_ids[i] == chain_ids[j]:
                    intra_distances.append(r)
                else:
                    inter_distances.append(r)

    # Compute histograms and normalize
    # ... (similar to above)

    return r, g_inter, g_intra
```

9.6 Static Structure Factor

The static structure factor $S(q)$ is measurable by scattering experiments:

$$\boxed{S(q) = \frac{1}{N}\langle \sum_{i,j} \exp[i\mathbf{q} \cdot (\mathbf{r}_i - \mathbf{r}_j)] \rangle} \tag{9.14}$$

For isotropic systems:

$$S(q) = 1 + 4\pi\rho \int_0^\infty r^2 [g(r) - 1] \frac{\sin(qr)}{qr} \, \mathrm{d}r \tag{9.15}$$

```python
def compute_structure_factor(positions, box_size, q_max=10, n_q=100):
    """
    Compute static structure factor S(q).
    """
    import numpy as np

    n_particles = len(positions)

    # q vectors compatible with periodic boundaries
    q_values = np.linspace(0.1, q_max, n_q)
    S_q = np.zeros(n_q)

    for q_idx, q in enumerate(q_values):
```

```
        # Average over q directions
        n_directions = 50
        S_sum = 0

        for _ in range(n_directions):
            # Random q direction
            theta = np.arccos(2 * np.random.rand() - 1)
            phi = 2 * np.pi * np.random.rand()
            q_vec = q * np.array([
                np.sin(theta) * np.cos(phi),
                np.sin(theta) * np.sin(phi),
                np.cos(theta)
            ])

            # Compute S(q)
            rho_q = np.sum(np.exp(1j * np.dot(positions, q_vec)))
            S_sum += np.abs(rho_q)**2 / n_particles

        S_q[q_idx] = S_sum / n_directions

    return q_values, S_q
```

9.7 Form Factor

The single-chain form factor:

$$P(q) = \frac{1}{N^2} \langle \sum_{i,j \in \text{chain}} \exp[i\mathbf{q} \cdot (\mathbf{r}_i - \mathbf{r}_j)] \rangle \tag{9.16}$$

For Gaussian chains (Debye function):

$$\boxed{P(q) = \frac{2}{x^2}(e^{-x} - 1 + x) \quad \text{where } x = q^2 R_g^2} \tag{9.17}$$

```
def compute_form_factor(positions, q_values):
    """
    Compute single-chain form factor P(q).
    """
    import numpy as np

    n_chains, n_beads, _ = positions.shape
    P_q = np.zeros(len(q_values))

    for q_idx, q in enumerate(q_values):
        P_sum = 0

        for chain_idx in range(n_chains):
            chain_pos = positions[chain_idx]

            # Average over q directions
            for _ in range(20):
                theta = np.arccos(2 * np.random.rand() - 1)
                phi = 2 * np.pi * np.random.rand()
                q_vec = q * np.array([
                    np.sin(theta) * np.cos(phi),
                    np.sin(theta) * np.sin(phi),
```

```
                    np.cos(theta)
                ])

                phases = np.exp(1j * np.dot(chain_pos, q_vec))
                rho_q = np.sum(phases)
                P_sum += np.abs(rho_q)**2 / n_beads**2

        P_q[q_idx] = P_sum / (n_chains * 20)

    return q_values, P_q


def debye_function(q, Rg):
    """Theoretical Debye form factor for Gaussian chain."""
    import numpy as np
    x = (q * Rg)**2
    return 2 * (np.exp(-x) - 1 + x) / x**2
```

9.8 Density Profiles and Interfaces

9.8.1 Local Density Profile

```
def compute_density_profile(positions, box_dims, axis=2, n_bins=100):
    """
    Compute density profile along specified axis.
    """
    import numpy as np

    coords = positions[:, axis]

    hist, bin_edges = np.histogram(coords, bins=n_bins,
                                   range=(0, box_dims[axis]))
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

    # Convert to density
    bin_volume = (box_dims[0] * box_dims[1] *
                  (bin_edges[1] - bin_edges[0]))
    density = hist / bin_volume

    return bin_centers, density
```

9.9 Summary of Key Metrics

Metric	Symbol	Physical Meaning
End-to-end distance	R_{ee}	Chain extension
Radius of gyration	R_g	Overall chain size
Characteristic ratio	C_∞	Chain stiffness
Shape anisotropy	κ^2	Deviation from spherical
RDF	$g(r)$	Local packing
Structure factor	$S(q)$	Scattering intensity
Form factor	$P(q)$	Single-chain scattering

9.10 Exercises

1. Compute R_{g} and R_{ee} for your polymer simulation. Verify $\langle R_{ee}^2 \rangle / \langle R_{\mathrm{g}}^2 \rangle \approx 6$ for long chains.
2. Analyze the dihedral angle distribution. Calculate trans and gauche fractions.
3. Compute the form factor $P(q)$ and compare with the Debye function using your measured R_{g}.
4. Implement a function to compute the intra- and inter-molecular $g(r)$ separately.
5. Calculate shape parameters and characterize how chain shape changes with molecular weight.

9.11 Further Reading

- Rubinstein, M. & Colby, R.H. *Polymer Physics*, Chapter 2
- Strobl, G. *The Physics of Polymers*, Chapter 3
- Higgins, J.S. & Benoit, H. *Polymers and Neutron Scattering*

Chapter 10

Dynamic Properties

10.1 Introduction to Polymer Dynamics

Polymer dynamics encompasses the time-dependent behavior of chains, from local segmental motion to global chain diffusion. Understanding dynamics is essential for predicting processing behavior, transport properties, and mechanical response.

Definition 10.1 (Relaxation Time)**.** *A relaxation time τ characterizes the time scale over which a system returns to equilibrium after a perturbation. Polymers exhibit a spectrum of relaxation times spanning many orders of magnitude.*

10.2 Mean Square Displacement

10.2.1 Definition and Regimes

The mean square displacement (MSD) measures particle mobility:

$$\boxed{\text{MSD}(t) = \langle |\mathbf{r}(t) - \mathbf{r}(0)|^2 \rangle} \tag{10.1}$$

For polymer melts, the MSD shows distinct regimes:

Regime	Time Range	MSD scaling	Physics
Ballistic	$t < \tau_0$	$\sim t^2$	Free particle motion
Cage/local	$\tau_0 < t < \tau_e$	$\sim t^{1/2}$	Rouse-like, local
Constrained Rouse	$\tau_e < t < \tau_R$	$\sim t^{1/4}$	Tube constraints
Reptation	$\tau_R < t < \tau_d$	$\sim t^{1/2}$	Curvilinear diffusion
Diffusive	$t > \tau_d$	$\sim t^1$	Free diffusion

```
def compute_msd(trajectory, particle_type='all', unwrap=True):
    """
    Compute mean square displacement from trajectory.

    Args:
        trajectory: List of frames with positions
        particle_type: Which particles to include
        unwrap: Whether to unwrap periodic boundaries

    Returns:
        time array, MSD array
    """
```

```
    import numpy as np

    n_frames = len(trajectory)
    n_particles = len(trajectory[0].positions)

    # Get positions (unwrapped for MSD)
    if unwrap:
        positions = unwrap_trajectory(trajectory)
    else:
        positions = np.array([frame.positions for frame in trajectory])

    # Compute MSD using multiple time origins
    max_lag = n_frames // 2
    msd = np.zeros(max_lag)
    counts = np.zeros(max_lag)

    for t0 in range(n_frames - max_lag):
        for lag in range(max_lag):
            disp = positions[t0 + lag] - positions[t0]
            msd[lag] += np.mean(np.sum(disp**2, axis=1))
            counts[lag] += 1

    msd /= counts

    # Time array
    dt = trajectory[1].time - trajectory[0].time
    time = np.arange(max_lag) * dt

    return time, msd


def unwrap_trajectory(trajectory):
    """
    Unwrap periodic boundary conditions for MSD calculation.
    """
    import numpy as np

    n_frames = len(trajectory)
    n_particles = len(trajectory[0].positions)
    box = trajectory[0].box

    positions = np.zeros((n_frames, n_particles, 3))
    positions[0] = trajectory[0].positions.copy()

    for i in range(1, n_frames):
        delta = trajectory[i].positions - trajectory[i-1].positions

        # Correct for PBC jumps
        delta -= box * np.round(delta / box)

        positions[i] = positions[i-1] + delta

    return positions
```

10.2.2 Center of Mass MSD

For diffusion coefficient measurement, use chain center of mass:

```python
def compute_com_msd(trajectory, chain_indices):
    """
    Compute center-of-mass MSD for chains.
    """
    import numpy as np

    n_frames = len(trajectory)
    n_chains = len(chain_indices)

    # Compute COM positions for each frame
    com_positions = np.zeros((n_frames, n_chains, 3))

    for frame_idx, frame in enumerate(trajectory):
        positions = unwrap_trajectory([frame])[0]
        for chain_idx, indices in enumerate(chain_indices):
            com_positions[frame_idx, chain_idx] = np.mean(
                positions[indices], axis=0
            )

    # Compute MSD
    max_lag = n_frames // 2
    msd = np.zeros(max_lag)

    for lag in range(max_lag):
        disp = com_positions[lag:] - com_positions[:n_frames-lag]
        msd[lag] = np.mean(np.sum(disp**2, axis=2))

    dt = trajectory[1].time - trajectory[0].time
    time = np.arange(max_lag) * dt

    return time, msd
```

10.3 Diffusion Coefficient

10.3.1 Einstein Relation

The diffusion coefficient is extracted from long-time MSD:

$$\boxed{D = \lim_{t \to \infty} \frac{\text{MSD}(t)}{6t}} \tag{10.2}$$

In practice, fit the linear regime:

$$\text{MSD}(t) = 6Dt + C \tag{10.3}$$

```python
def compute_diffusion_coefficient(time, msd, t_min=None, t_max=None):
    """
    Compute diffusion coefficient from MSD.

    Args:
        time: Time array
        msd: MSD array
        t_min, t_max: Fitting range (diffusive regime)

    Returns:
```

```
        D, uncertainty
    """
    import numpy as np
    from scipy.optimize import curve_fit

    if t_min is None:
        t_min = time[-1] / 4
    if t_max is None:
        t_max = time[-1] / 2

    # Select diffusive regime
    mask = (time >= t_min) & (time <= t_max)
    t_fit = time[mask]
    msd_fit = msd[mask]

    # Linear fit
    def linear(t, D, C):
        return 6 * D * t + C

    popt, pcov = curve_fit(linear, t_fit, msd_fit)
    D = popt[0]
    D_err = np.sqrt(pcov[0, 0])

    return D, D_err
```

10.3.2 Green-Kubo Relation

Alternative diffusion coefficient from velocity autocorrelation:

$$\boxed{D = \frac{1}{3} \int_0^\infty \langle \mathbf{v}(0) \cdot \mathbf{v}(t) \rangle \, \mathrm{d}t} \tag{10.4}$$

```
def compute_diffusion_gk(trajectory):
    """
    Compute diffusion coefficient via Green-Kubo.
    """
    import numpy as np

    n_frames = len(trajectory)
    n_particles = len(trajectory[0].velocities)

    # Extract velocities
    velocities = np.array([frame.velocities for frame in trajectory])

    # Compute VACF
    max_lag = n_frames // 4
    vacf = np.zeros(max_lag)

    for lag in range(max_lag):
        v_dot = np.sum(
            velocities[:n_frames-lag] * velocities[lag:],
            axis=2
        )
        vacf[lag] = np.mean(v_dot)

    # Integrate
    dt = trajectory[1].time - trajectory[0].time
```

```
    D = np.trapz(vacf, dx=dt) / 3

    return D, vacf
```

10.3.3 Molecular Weight Dependence

Regime	D scaling	Model
Unentangled	$D \sim N^{-1}$	Rouse
Entangled	$D \sim N^{-2.0}$ to $N^{-2.3}$	Reptation

10.4 Relaxation Times

10.4.1 End-to-End Vector Relaxation

The end-to-end vector autocorrelation:

$$C_{ee}(t) = \frac{\langle \mathbf{R}_{ee}(0) \cdot \mathbf{R}_{ee}(t) \rangle}{\langle R_{ee}^2 \rangle} \tag{10.5}$$

```
def compute_ree_relaxation(trajectory, chain_indices):
    """
    Compute end-to-end vector autocorrelation function.
    """
    import numpy as np

    n_frames = len(trajectory)
    n_chains = len(chain_indices)

    # Compute Ree vectors for each frame
    Ree = np.zeros((n_frames, n_chains, 3))

    for frame_idx, frame in enumerate(trajectory):
        for chain_idx, indices in enumerate(chain_indices):
            Ree[frame_idx, chain_idx] = (
                frame.positions[indices[-1]] -
                frame.positions[indices[0]]
            )

    # Autocorrelation
    max_lag = n_frames // 2
    C_ee = np.zeros(max_lag)

    Ree_sq = np.mean(np.sum(Ree**2, axis=2))

    for lag in range(max_lag):
        dot_products = np.sum(Ree[:n_frames-lag] * Ree[lag:], axis=2)
        C_ee[lag] = np.mean(dot_products) / Ree_sq

    dt = trajectory[1].time - trajectory[0].time
    time = np.arange(max_lag) * dt

    # Extract relaxation time (where C = 1/e)
    idx = np.where(C_ee < 1/np.e)[0]
    tau_ee = time[idx[0]] if len(idx) > 0 else time[-1]

    return time, C_ee, tau_ee
```

10.4.2 Rouse Mode Analysis

```
def compute_rouse_modes(trajectory, chain_indices, n_modes=10):
    """
    Compute Rouse mode autocorrelation functions.

    X_p = (1/N) * sum_n R_n * cos(p*pi*(n-0.5)/N)
    """
    import numpy as np

    n_frames = len(trajectory)
    n_chains = len(chain_indices)
    n_beads = len(chain_indices[0])

    # Compute Rouse modes
    modes = np.zeros((n_frames, n_chains, n_modes, 3))

    for frame_idx, frame in enumerate(trajectory):
        for chain_idx, indices in enumerate(chain_indices):
            chain_pos = frame.positions[indices]

            for p in range(1, n_modes + 1):
                for n in range(n_beads):
                    coeff = np.cos(p * np.pi * (n + 0.5) / n_beads)
                    modes[frame_idx, chain_idx, p-1] += chain_pos[n] *
   coeff
                modes[frame_idx, chain_idx, p-1] /= n_beads

    # Compute ACF for each mode
    tau_p = np.zeros(n_modes)
    max_lag = n_frames // 2

    for p in range(n_modes):
        mode_acf = np.zeros(max_lag)
        mode_sq = np.mean(np.sum(modes[:, :, p, :]**2, axis=2))

        for lag in range(max_lag):
            dot = np.sum(modes[:n_frames-lag, :, p, :] *
                         modes[lag:, :, p, :], axis=2)
            mode_acf[lag] = np.mean(dot) / mode_sq

        # Find relaxation time
        idx = np.where(mode_acf < 1/np.e)[0]
        dt = trajectory[1].time - trajectory[0].time
        tau_p[p] = idx[0] * dt if len(idx) > 0 else max_lag * dt

    # Rouse prediction: tau_p = tau_1 / p^2
    return tau_p
```

10.4.3 Segmental Relaxation

Local backbone dynamics from bond vector autocorrelation:

```
def compute_segmental_relaxation(trajectory, bond_pairs):
    """
    Compute bond vector autocorrelation for segmental dynamics.

    P2(t) = (3*<cos^2(theta)> - 1) / 2
```

```
    """
    import numpy as np

    n_frames = len(trajectory)
    n_bonds = len(bond_pairs)

    # Compute bond vectors
    bond_vectors = np.zeros((n_frames, n_bonds, 3))

    for frame_idx, frame in enumerate(trajectory):
        for bond_idx, (i, j) in enumerate(bond_pairs):
            bond_vectors[frame_idx, bond_idx] = (
                frame.positions[j] - frame.positions[i]
            )
            # Normalize
            bond_vectors[frame_idx, bond_idx] /= np.linalg.norm(
                bond_vectors[frame_idx, bond_idx]
            )

    # Compute P2 correlation
    max_lag = n_frames // 2
    P2 = np.zeros(max_lag)

    for lag in range(max_lag):
        cos_theta = np.sum(
            bond_vectors[:n_frames-lag] * bond_vectors[lag:],
            axis=2
        )
        P2[lag] = np.mean(0.5 * (3 * cos_theta**2 - 1))

    dt = trajectory[1].time - trajectory[0].time
    time = np.arange(max_lag) * dt

    # Segmental relaxation time
    idx = np.where(P2 < 1/np.e)[0]
    tau_seg = time[idx[0]] if len(idx) > 0 else time[-1]

    return time, P2, tau_seg
```

10.5 Viscosity from MD

10.5.1 Green-Kubo Method

$$\eta = \frac{V}{k_B T} \int_0^\infty \langle \sigma_{\alpha\beta}(0) \sigma_{\alpha\beta}(t) \rangle \, dt \tag{10.6}$$

```
def compute_viscosity(trajectory, temperature, volume):
    """
    Compute shear viscosity via Green-Kubo.
    """
    import numpy as np

    kB = 1.380649e-23  # J/K

    n_frames = len(trajectory)

    # Extract off-diagonal stress components
```

```
    stress_xy = np.array([frame.stress[0, 1] for frame in trajectory])
    stress_xz = np.array([frame.stress[0, 2] for frame in trajectory])
    stress_yz = np.array([frame.stress[1, 2] for frame in trajectory])

    # Compute stress ACF
    max_lag = n_frames // 4

    def stress_acf(stress):
        acf = np.zeros(max_lag)
        for lag in range(max_lag):
            acf[lag] = np.mean(stress[:n_frames-lag] * stress[lag:])
        return acf

    acf_xy = stress_acf(stress_xy)
    acf_xz = stress_acf(stress_xz)
    acf_yz = stress_acf(stress_yz)

    # Average
    acf_avg = (acf_xy + acf_xz + acf_yz) / 3

    # Integrate (running integral for plateau)
    dt = trajectory[1].time - trajectory[0].time
    running_integral = np.cumsum(acf_avg) * dt

    # Prefactor
    prefactor = volume / (kB * temperature)

    eta_t = prefactor * running_integral

    # Viscosity from plateau
    # Find where running integral plateaus
    plateau_start = max_lag // 2
    eta = np.mean(eta_t[plateau_start:])

    return eta, eta_t
```

10.5.2 NEMD Approach

```
# LAMMPS NEMD viscosity measurement
"""
# Apply shear
fix shear all deform 1 xy erate 0.001 remap v

# Compute stress
compute stress all pressure NULL

# Average shear stress
variable sxy equal -pxy
fix avgstress all ave/time 100 10 1000 v_sxy file stress.dat

# Viscosity = <sigma_xy> / shear_rate
"""
```

10.6 Temperature Dependence

10.6.1 Arrhenius Behavior

For dynamics above $T_g + 100$ K:

$$\tau(T) = \tau_0 \exp\left(\frac{E_a}{k_B T}\right) \tag{10.7}$$

10.6.2 VFT/WLF Behavior

Near T_g:

$$\boxed{\log \frac{\tau(T)}{\tau(T_{\text{ref}})} = \frac{-C_1(T - T_{\text{ref}})}{C_2 + (T - T_{\text{ref}})}} \tag{10.8}$$

```python
def fit_wlf(temperatures, relaxation_times, T_ref=None):
    """
    Fit WLF equation to relaxation time data.
    """
    import numpy as np
    from scipy.optimize import curve_fit

    if T_ref is None:
        T_ref = min(temperatures)

    tau_ref = np.interp(T_ref, temperatures, relaxation_times)

    def wlf(T, C1, C2):
        return tau_ref * 10**(-C1 * (T - T_ref) / (C2 + T - T_ref))

    popt, pcov = curve_fit(wlf, temperatures, relaxation_times,
                           p0=[17.4, 51.6])

    C1, C2 = popt

    return C1, C2, T_ref
```

10.7 Summary of Scaling Relations

Property	Symbol	Rouse	Reptation
Diffusion	D	N^{-1}	N^{-2}
Viscosity	η	N^{1}	$N^{3.4}$
Longest relaxation	τ_1	N^{2}	N^{3}
Rouse modes	τ_p	τ_1/p^2	—

10.8 Exercises

1. Compute the MSD for your polymer system. Identify the different dynamical regimes.
2. Extract the diffusion coefficient from both MSD (Einstein) and VACF (Green-Kubo). Compare results.
3. Compute Rouse mode relaxation times and verify $\tau_p \sim p^{-2}$.

4. Measure viscosity using Green-Kubo. How long must you run to converge?
5. Determine the temperature dependence of segmental relaxation and fit to WLF.

10.9 Further Reading

- Doi, M. & Edwards, S.F. *The Theory of Polymer Dynamics*
- Rubinstein, M. & Colby, R.H. *Polymer Physics*, Chapters 7–9
- Allen, M.P. & Tildesley, D.J. *Computer Simulation of Liquids*

Chapter 11

Mechanical Properties from Simulation

11.1 Introduction to Mechanical Properties

Mechanical properties describe a material's response to applied forces. MD simulations can predict elastic moduli, yield behavior, and failure mechanisms at the molecular level, providing insights not accessible experimentally.

Definition 11.1 (Mechanical Response)**.** *The relationship between applied stress σ and resulting strain ϵ characterizes mechanical behavior. This relationship depends on temperature, strain rate, and molecular structure.*

11.2 Stress and Strain Fundamentals

11.2.1 Stress Tensor

The microscopic stress tensor (virial stress):

$$\boxed{\sigma_{\alpha\beta} = -\frac{1}{V}\left[\sum_i m_i v_{i\alpha} v_{i\beta} + \sum_{i<j} r_{ij\alpha} F_{ij\beta}\right]} \tag{11.1}$$

where the first term is kinetic and the second is the virial contribution from interatomic forces.

```
def compute_stress_tensor(positions, velocities, forces, masses,
                          box_volume, pair_forces=None):
    """
    Compute the stress tensor from atomic quantities.

    Args:
        positions: Particle positions
        velocities: Particle velocities
        forces: Forces on particles
        masses: Particle masses
        box_volume: System volume
        pair_forces: Pairwise force contributions (for virial)

    Returns:
        3x3 stress tensor in pressure units
    """
    import numpy as np
```

```
    n_particles = len(positions)

    # Kinetic contribution
    sigma_kinetic = np.zeros((3, 3))
    for i in range(n_particles):
        sigma_kinetic += masses[i] * np.outer(velocities[i], velocities[i])

    # Virial contribution (requires pairwise forces)
    sigma_virial = np.zeros((3, 3))
    if pair_forces is not None:
        for i, j, f_ij in pair_forces:
            r_ij = positions[j] - positions[i]
            sigma_virial += np.outer(r_ij, f_ij)

    # Total stress
    sigma = -(sigma_kinetic + sigma_virial) / box_volume

    return sigma
```

11.2.2 Strain Measures

Engineering strain:

$$\epsilon = \frac{L - L_0}{L_0} = \frac{\Delta L}{L_0} \tag{11.2}$$

True (logarithmic) strain:

$$\epsilon_{\text{true}} = \ln\left(\frac{L}{L_0}\right) = \ln(1 + \epsilon) \tag{11.3}$$

Strain tensor for small deformations:

$$\epsilon_{\alpha\beta} = \frac{1}{2}\left(\frac{\partial u_\alpha}{\partial x_\beta} + \frac{\partial u_\beta}{\partial x_\alpha}\right) \tag{11.4}$$

11.3 Elastic Moduli

11.3.1 Young's Modulus

Young's modulus from uniaxial tension/compression:

$$\boxed{E = \frac{\sigma_{11}}{\epsilon_{11}} \quad \text{(uniaxial stress)}} \tag{11.5}$$

11.3.2 Bulk Modulus

Bulk modulus from volumetric compression:

$$K = -V\frac{\mathrm{d}P}{\mathrm{d}V} = \frac{\sigma_{\text{hydrostatic}}}{\epsilon_{\text{volumetric}}} \tag{11.6}$$

11.3.3 Shear Modulus

Shear modulus from simple shear:

$$G = \frac{\sigma_{12}}{\gamma_{12}} \tag{11.7}$$

11.3.4 Poisson's Ratio

$$\nu = -\frac{\epsilon_{\text{transverse}}}{\epsilon_{\text{axial}}} = -\frac{\epsilon_{22}}{\epsilon_{11}} \tag{11.8}$$

Relationships (isotropic materials):

$$E = 2G(1+\nu) = 3K(1-2\nu) \tag{11.9}$$

Polymer	E (GPa)	G (GPa)	K (GPa)	ν
Polyethylene (HDPE)	0.8–1.2	0.3–0.4	2–4	0.42
Polystyrene	3.0–3.5	1.2–1.4	3–5	0.33
PMMA	2.5–3.2	1.0–1.2	4–6	0.35
Polycarbonate	2.3–2.5	0.8–0.9	3–4	0.37

11.4 MD Methods for Mechanical Properties

11.4.1 Direct Deformation

Apply controlled deformation and measure stress response:

```
# LAMMPS uniaxial tension simulation
"""
# Fix one dimension, rescale others for constant volume
# or use constant pressure in transverse directions

# Uniaxial strain in z-direction
variable strain equal 0.003  # 0.3% per step
variable erate equal 1e8     # strain rate in 1/s

fix deform all deform 1 z erate ${erate} remap x

# Compute stress
compute stress all pressure NULL

# Output stress-strain
variable szz equal -pzz
variable eps equal (lz-v_lz0)/v_lz0

fix output all ave/time 100 10 1000 v_eps v_szz file stress_strain.dat

# Run deformation
run 100000
"""
```

```
def perform_uniaxial_tension(system, max_strain=0.1, strain_rate=1e8):
    """
    Perform uniaxial tension simulation.

    Args:
        system: MD system object
        max_strain: Maximum engineering strain
        strain_rate: Strain rate in 1/s

    Returns:
        strain, stress arrays
    """
```

```
    import numpy as np

    L0 = system.box[2]  # Initial length in z
    dt = system.timestep

    # Time to reach max strain
    total_time = max_strain / strain_rate
    n_steps = int(total_time / dt)

    strain_data = []
    stress_data = []

    for step in range(n_steps):
        # Current strain
        eps = strain_rate * step * dt

        # Deform box
        new_Lz = L0 * (1 + eps)
        system.box[2] = new_Lz

        # Rescale positions
        scale_factor = new_Lz / system.box_old[2]
        system.positions[:, 2] *= scale_factor

        # Run short equilibration
        system.run(steps=100, ensemble='nvt')

        # Measure stress
        sigma = system.compute_stress()

        strain_data.append(eps)
        stress_data.append(sigma[2, 2])

    return np.array(strain_data), np.array(stress_data)
```

11.4.2 Fluctuation Method

Elastic constants from stress fluctuations in NVT:

$$\boxed{C_{\alpha\beta\gamma\delta} = \frac{V}{k_B T}\left[\langle\sigma_{\alpha\beta}\sigma_{\gamma\delta}\rangle - \langle\sigma_{\alpha\beta}\rangle\langle\sigma_{\gamma\delta}\rangle\right] + C^{\text{Born}}} \tag{11.10}$$

The Born term accounts for affine deformation:

$$C^{\text{Born}}_{\alpha\beta\gamma\delta} = \frac{1}{V}\langle\sum_{i<j}\frac{\partial^2 U}{\partial r_{ij\alpha}\partial r_{ij\gamma}} r_{ij\beta} r_{ij\delta}\rangle \tag{11.11}$$

```
def compute_elastic_constants_fluctuation(trajectory, temperature, volume):
    """
    Compute elastic constants from stress fluctuations.
    """
    import numpy as np

    kB = 1.380649e-23
    n_frames = len(trajectory)

    # Collect stress tensors
```

```
    stress = np.array([frame.stress for frame in trajectory])

    # Compute fluctuation contribution
    # C_ijkl = (V/kT) * [<s_ij s_kl> - <s_ij><s_kl>]
    C = np.zeros((6, 6))  # Voigt notation

    # Mapping: 11->0, 22->1, 33->2, 23->3, 13->4, 12->5
    voigt_map = {
        (0,0): 0, (1,1): 1, (2,2): 2,
        (1,2): 3, (2,1): 3,
        (0,2): 4, (2,0): 4,
        (0,1): 5, (1,0): 5
    }

    for i in range(3):
        for j in range(i, 3):
            vi = voigt_map[(i,j)]
            s_ij = stress[:, i, j]

            for k in range(3):
                for l in range(k, 3):
                    vk = voigt_map[(k,l)]
                    s_kl = stress[:, k, l]

                    C[vi, vk] = (volume / (kB * temperature)) * (
                        np.mean(s_ij * s_kl) - np.mean(s_ij) * np.mean(s_kl
)
                    )

    # Note: This gives fluctuation contribution only
    # Born term needs to be added for complete result

    return C
```

11.4.3 Small Oscillatory Deformation

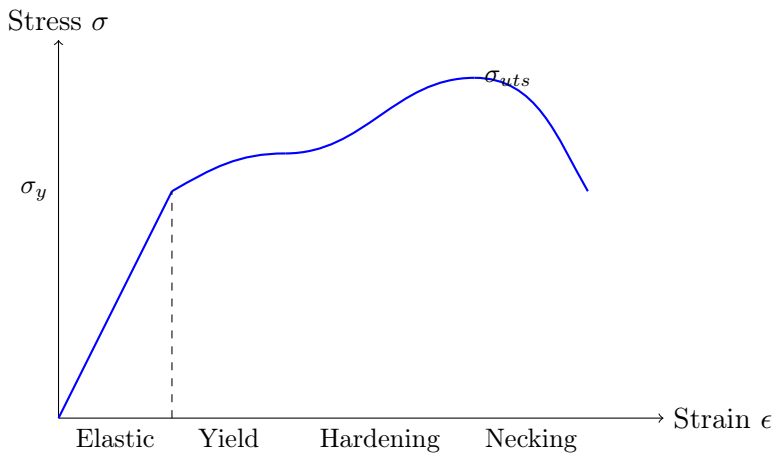
Apply small sinusoidal strain and measure stress response:

```
def compute_complex_modulus(system, frequency, strain_amplitude=0.01):
    """
    Compute complex modulus from oscillatory shear.

    G* = G' + iG''
    """
    import numpy as np

    omega = 2 * np.pi * frequency
    period = 1 / frequency
    n_cycles = 5
    dt = system.timestep

    n_steps_per_cycle = int(period / dt)
    total_steps = n_cycles * n_steps_per_cycle

    time = np.arange(total_steps) * dt
    strain = strain_amplitude * np.sin(omega * time)
    stress = np.zeros(total_steps)
```

```
    for step in range(total_steps):
        # Apply strain
        gamma = strain[step]
        system.apply_shear_strain(gamma)

        # Run one timestep
        system.run(steps=1)

        # Measure stress
        stress[step] = system.compute_stress()[0, 1]

    # Fit to sigma = gamma_0 * (G' sin(wt) + G'' cos(wt))
    # Fourier analysis
    stress_fft = np.fft.fft(stress)
    strain_fft = np.fft.fft(strain)

    freq_idx = n_cycles  # Index of fundamental frequency
    G_complex = stress_fft[freq_idx] / strain_fft[freq_idx]

    G_prime = np.real(G_complex)
    G_double_prime = np.imag(G_complex)

    return G_prime, G_double_prime
```

11.5 Stress-Strain Behavior

11.5.1 Regions of the Stress-Strain Curve



11.5.2 Yield Stress Detection

```
def detect_yield_point(strain, stress, method='offset'):
    """
    Detect yield point from stress-strain data.

    Methods:
        'offset': 0.2% offset method
        'tangent': Intersection of tangents
        'max_curvature': Maximum curvature point
    """
    import numpy as np
```

```
    from scipy.signal import savgol_filter

    if method == 'offset':
        # Find elastic modulus from initial slope
        n_elastic = len(strain) // 10
        E = np.polyfit(strain[:n_elastic], stress[:n_elastic], 1)[0]

        # Offset line: sigma = E * (epsilon - 0.002)
        offset_stress = E * (strain - 0.002)

        # Find intersection
        diff = stress - offset_stress
        idx = np.where(diff < 0)[0]
        yield_idx = idx[0] if len(idx) > 0 else len(strain) - 1

    elif method == 'max_curvature':
        # Smooth data
        stress_smooth = savgol_filter(stress, 11, 3)

        # Compute curvature
        ds = np.gradient(stress_smooth, strain)
        d2s = np.gradient(ds, strain)
        curvature = np.abs(d2s) / (1 + ds**2)**1.5

        yield_idx = np.argmax(curvature)

    return strain[yield_idx], stress[yield_idx]
```

11.6 Polymer-Specific Mechanical Behavior

11.6.1 Rubber Elasticity

For crosslinked rubbers above T_g:

$$\boxed{\sigma = G(\lambda - \lambda^{-2}) = \frac{\rho k_B T}{M_c}(\lambda - \lambda^{-2})} \tag{11.12}$$

where $\lambda = L/L_0$ is the stretch ratio and M_c is the molecular weight between crosslinks.

```
def fit_rubber_elasticity(strain, stress, temperature, density):
    """
    Fit neo-Hookean rubber elasticity model.

    sigma = G * (lambda - lambda^-2)
    """
    import numpy as np
    from scipy.optimize import curve_fit

    # Convert to stretch ratio
    lamda = 1 + strain

    def neo_hookean(lam, G):
        return G * (lam - lam**(-2))

    popt, _ = curve_fit(neo_hookean, lamda, stress, p0=[1e6])
    G = popt[0]
```

```
    # Molecular weight between crosslinks
    kB = 1.380649e-23
    NA = 6.022e23
    Mc = density * kB * temperature * NA / G

    return G, Mc
```

11.6.2 Crazing and Shear Banding

Glassy polymers may exhibit localized deformation:

```
def detect_localized_deformation(positions, initial_positions, n_slices=50)
    :
    """
    Detect strain localization (crazes, shear bands).
    """
    import numpy as np

    # Compute local strain in z-direction
    z_initial = initial_positions[:, 2]
    z_final = positions[:, 2]

    z_min, z_max = np.min(z_initial), np.max(z_initial)
    slice_edges = np.linspace(z_min, z_max, n_slices + 1)

    local_strain = np.zeros(n_slices)

    for i in range(n_slices):
        mask = (z_initial >= slice_edges[i]) & (z_initial < slice_edges[i
    +1])
        if np.sum(mask) > 0:
            dz_local = np.mean(z_final[mask]) - np.mean(z_initial[mask])
            dz0_local = slice_edges[i+1] - slice_edges[i]
            local_strain[i] = dz_local / dz0_local

    # Localization detected if strain variance is high
    strain_variance = np.var(local_strain)
    is_localized = strain_variance > 0.01

    return local_strain, is_localized
```

11.6.3 Strain Hardening

Polymers exhibit strain hardening at large strains due to chain orientation:

```
def compute_chain_orientation(positions, chain_indices, deformation_axis=2)
    :
    """
    Compute chain orientation parameter during deformation.

    S = (3*<cos^2(theta)> - 1) / 2

    S = 0: random
    S = 1: perfectly aligned with deformation axis
    S = -0.5: perpendicular to deformation axis
    """
    import numpy as np
```

```
    n_chains = len(chain_indices)
    S_values = []

    for indices in chain_indices:
        # End-to-end vector
        Ree = positions[indices[-1]] - positions[indices[0]]
        Ree_norm = Ree / np.linalg.norm(Ree)

        # Angle with deformation axis
        cos_theta = Ree_norm[deformation_axis]

        S_values.append(0.5 * (3 * cos_theta**2 - 1))

    S = np.mean(S_values)

    return S
```

11.7 Temperature and Rate Effects

11.7.1 Time-Temperature Superposition

Mechanical properties at different temperatures and rates can be superposed:

```
def apply_tts_to_mechanical_data(T, strain_rate, T_ref=300):
    """
    Apply time-temperature superposition shift factor.

    Shifted strain rate: strain_rate_shifted = strain_rate * aT
    """
    import numpy as np

    # WLF parameters (example values)
    C1, C2 = 17.4, 51.6

    log_aT = -C1 * (T - T_ref) / (C2 + T - T_ref)
    aT = 10**log_aT

    shifted_rate = strain_rate * aT

    return shifted_rate, aT
```

11.7.2 Strain Rate Dependence

```
def study_strain_rate_dependence(system, strain_rates):
    """
    Study mechanical properties at different strain rates.
    """
    results = []

    for rate in strain_rates:
        strain, stress = perform_uniaxial_tension(
            system.copy(), max_strain=0.3, strain_rate=rate
        )

        # Extract properties
```

```
        E = compute_youngs_modulus(strain[:50], stress[:50])
        sigma_y, eps_y = detect_yield_point(strain, stress)

        results.append({
            'strain_rate': rate,
            'E': E,
            'yield_stress': sigma_y,
            'yield_strain': eps_y
        })

    return results
```

11.8 Simulation Protocol for Mechanical Testing

```
def mechanical_testing_protocol(system_file, temperature=300):
    """
    Complete protocol for mechanical property determination.
    """
    # Load and equilibrate system
    system = load_system(system_file)
    system.run_equilibration(steps=1000000, temp=temperature)

    results = {}

    # 1. Bulk modulus from NPT fluctuations
    system.run_npt(steps=500000, temp=temperature, press=1.0)
    results['bulk_modulus'] = compute_bulk_modulus_from_fluctuations(
        system.trajectory
    )

    # 2. Elastic constants from stress fluctuations
    system.run_nvt(steps=500000, temp=temperature)
    results['C_matrix'] = compute_elastic_constants_fluctuation(
        system.trajectory, temperature, system.volume
    )

    # 3. Uniaxial tension
    strain, stress = perform_uniaxial_tension(
        system, max_strain=0.5, strain_rate=1e8
    )
    results['stress_strain'] = (strain, stress)
    results['youngs_modulus'] = compute_youngs_modulus(strain[:100], stress
    [:100])
    results['yield_point'] = detect_yield_point(strain, stress)

    # 4. Simple shear
    gamma, tau = perform_simple_shear(system, max_shear=0.3)
    results['shear_modulus'] = compute_shear_modulus(gamma[:100], tau
    [:100])

    return results
```

11.9 Summary

Property	MD Method
Young's modulus E	Uniaxial tension, fluctuation method
Shear modulus G	Simple shear, oscillatory shear
Bulk modulus K	NPT volume fluctuations
Poisson's ratio ν	Uniaxial tension with transverse measurement
Yield stress	Stress-strain curve analysis

11.10 Exercises

1. Perform uniaxial tension on an amorphous polymer and extract Young's modulus.
2. Compare elastic constants from direct deformation vs. fluctuation methods.
3. Study the temperature dependence of yield stress from $T_g - 50$ K to $T_g + 50$ K.
4. For a rubber, verify the neo-Hookean model and extract crosslink density.
5. Investigate strain rate effects on mechanical properties.

11.11 Further Reading

- Ward, I.M. & Sweeney, J. *Mechanical Properties of Solid Polymers*
- Treloar, L.R.G. *The Physics of Rubber Elasticity*
- Frenkel, D. & Smit, B. *Understanding Molecular Simulation*

Part IV

Multiscale Methods

Chapter 12

Self-Consistent Field Theory

12.1 Introduction to Field-Theoretic Methods

Self-Consistent Field Theory (SCFT) provides a mesoscale description of polymer systems by replacing particle-particle interactions with particle-field interactions.

Definition 12.1 (SCFT)**.** *A mean-field theory that replaces the many-body polymer problem with single chains interacting with self-consistently determined fields representing the average environment.*

12.1.1 Why Field Theory?

Aspect	MD	SCFT
Degrees of freedom	$3N$ particles	Field on grid ($\sim 10^6$ points)
Time scales	ps–μs	Equilibrium only
System size	~ 100 nm	μm
Phase behavior	Requires long runs	Direct access
Morphology	Emerges dynamically	Predicted from free energy

12.2 Particle-to-Field Transformation

12.2.1 The Partition Function

For a system of n polymer chains in volume V:

$$\mathcal{Z} = \frac{1}{n!} \int \prod_{\alpha=1}^{n} \mathcal{D}[\mathbf{r}_\alpha(s)] \exp\left(-\frac{U[\{\mathbf{r}\}]}{k_\mathrm{B}T}\right) \tag{12.1}$$

where $\mathbf{r}_\alpha(s)$ is the position of segment s on chain α.

12.2.2 Hubbard-Stratonovich Transformation

The key insight is to introduce auxiliary fields. For a local interaction:

$$\exp\left(-\frac{\chi}{2}\int \mathrm{d}\mathbf{r}\, \hat{\rho}_A(\mathbf{r})\hat{\rho}_B(\mathbf{r})\right) \propto \int \mathcal{D}[W] \exp\left(-\frac{1}{2\chi}\int W^2 + i\int W(\hat{\rho}_A - \hat{\rho}_B)\right) \tag{12.2}$$

This transforms particle-particle interactions into particle-field interactions.

12.2.3 The Field-Theoretic Partition Function

After transformation:

$$\mathcal{Z} = \int \mathcal{D}[W_+]\mathcal{D}[W_-] \exp\left(-H[W_+, W_-]\right) \tag{12.3}$$

where:

- W_+: Pressure-like field (enforces incompressibility)
- W_-: Exchange field (drives phase separation)

12.3 The SCFT Equations

In the mean-field (saddle-point) approximation:

$$\frac{\delta H}{\delta W_+} = 0, \quad \frac{\delta H}{\delta W_-} = 0 \tag{12.4}$$

12.3.1 AB Diblock Copolymer

For a diblock with f fraction of A:

Field equations:

$$w_A(\mathbf{r}) = \chi N \phi_B(\mathbf{r}) + \xi(\mathbf{r}) \tag{12.5}$$
$$w_B(\mathbf{r}) = \chi N \phi_A(\mathbf{r}) + \xi(\mathbf{r}) \tag{12.6}$$

Density equations:

$$\phi_A(\mathbf{r}) = \frac{V}{Q} \int_0^f \mathrm{d}s\, q(\mathbf{r}, s) q^\dagger(\mathbf{r}, s) \tag{12.7}$$
$$\phi_B(\mathbf{r}) = \frac{V}{Q} \int_f^1 \mathrm{d}s\, q(\mathbf{r}, s) q^\dagger(\mathbf{r}, s) \tag{12.8}$$

Incompressibility:

$$\phi_A(\mathbf{r}) + \phi_B(\mathbf{r}) = 1 \tag{12.9}$$

12.3.2 Chain Propagators

The chain propagator $q(\mathbf{r}, s)$ satisfies a modified diffusion equation:

$$\boxed{\frac{\partial q}{\partial s} = \frac{b^2}{6}\nabla^2 q - w(\mathbf{r}) q} \tag{12.10}$$

with initial condition $q(\mathbf{r}, 0) = 1$.

Similarly, $q^\dagger(\mathbf{r}, s)$ propagates from the other end.

The partition function:

$$Q = \frac{1}{V} \int \mathrm{d}\mathbf{r}\, q(\mathbf{r}, 1) \tag{12.11}$$

12.4 Numerical Implementation

12.4.1 Pseudospectral Method

The diffusion equation can be solved efficiently using split-operator methods:

$$q(\mathbf{r}, s + \Delta s) \approx e^{-w\Delta s/2} e^{\Delta s \frac{b^2}{6}\nabla^2} e^{-w\Delta s/2} q(\mathbf{r}, s) \tag{12.12}$$

The Laplacian is diagonal in Fourier space: $\nabla^2 \to -k^2$.

```
import torch
import torch.fft as fft

class SCFTSolver:
    """Self-Consistent Field Theory solver for block copolymers."""

    def __init__(self, Nx: int, Lx: float, chi_N: float, f: float,
                 N_segments: int = 100):
        """
        Initialize SCFT solver.

        Args:
            Nx: Grid points per dimension
            Lx: Box size (in units of Rg)
            chi_N: Flory-Huggins parameter * chain length
            f: Volume fraction of A block
            N_segments: Discretization along chain
        """
        self.Nx = Nx
        self.Lx = Lx
        self.chi_N = chi_N
        self.f = f
        self.ds = 1.0 / N_segments

        # Real space grid
        self.dx = Lx / Nx
        x = torch.linspace(0, Lx - self.dx, Nx)
        self.X, self.Y, self.Z = torch.meshgrid(x, x, x, indexing='ij')

        # Fourier space grid
        kx = 2 * torch.pi * fft.fftfreq(Nx, self.dx)
        self.KX, self.KY, self.KZ = torch.meshgrid(kx, kx, kx, indexing='ij
    ')
        self.K2 = self.KX**2 + self.KY**2 + self.KZ**2

        # Diffusion propagator in k-space
        # For Gaussian chain: b^2/6 = Rg^2/N, using Rg units: coeff = 1
        self.exp_laplacian = torch.exp(-self.K2 * self.ds)

    def propagate_chain(self, w: torch.Tensor,
                        s_start: float, s_end: float) -> torch.Tensor:
        """
        Solve modified diffusion equation.

        dq/ds = Laplacian(q) - w*q

        Using split-operator method.
```

```
        Args:
            w: Field values on grid
            s_start: Starting contour position
            s_end: Ending contour position

        Returns:
            Propagator q(r, s_end)
        """
        # Initial condition
        q = torch.ones_like(w)

        n_steps = int(abs(s_end - s_start) / self.ds)
        half_w = torch.exp(-0.5 * w * self.ds)

        for _ in range(n_steps):
            # Split-operator: exp(-w*ds/2) * exp(Lap*ds) * exp(-w*ds/2)
            q = half_w * q
            q_k = fft.fftn(q)
            q_k = q_k * self.exp_laplacian
            q = fft.ifftn(q_k).real
            q = half_w * q

        return q

    def compute_densities(self, w_A: torch.Tensor,
                          w_B: torch.Tensor) -> tuple:
        """
        Compute segment densities from fields.

        Args:
            w_A: Field acting on A segments
            w_B: Field acting on B segments

        Returns:
            Tuple of (phi_A, phi_B, Q)
        """
        # Forward propagator (from A end)
        q_forward = [torch.ones_like(w_A)]
        n_A = int(self.f / self.ds)
        n_B = int((1 - self.f) / self.ds)

        # Propagate through A block
        q = q_forward[0]
        for _ in range(n_A):
            q = self.propagate_chain(w_A, 0, self.ds)
            q_forward.append(q.clone())

        # Propagate through B block
        for _ in range(n_B):
            q = self.propagate_chain(w_B, 0, self.ds)
            q_forward.append(q.clone())

        # Backward propagator
        q_backward = [torch.ones_like(w_B)]
        q = q_backward[0]

        for _ in range(n_B):
            q = self.propagate_chain(w_B, 0, self.ds)
```

```
            q_backward.append(q.clone())

        for _ in range(n_A):
            q = self.propagate_chain(w_A, 0, self.ds)
            q_backward.append(q.clone())

        q_backward = q_backward[::-1]

        # Partition function
        Q = q_forward[-1].mean()

        # Compute densities by integration
        V = self.Lx**3
        phi_A = torch.zeros_like(w_A)
        phi_B = torch.zeros_like(w_B)

        for i in range(n_A):
            phi_A += q_forward[i] * q_backward[i] * self.ds

        for i in range(n_A, n_A + n_B):
            phi_B += q_forward[i] * q_backward[i] * self.ds

        phi_A = phi_A * V / Q
        phi_B = phi_B * V / Q

        return phi_A, phi_B, Q

    def iterate(self, n_iter: int = 100,
                mix: float = 0.1) -> dict:
        """
        Run SCFT iteration.

        Args:
            n_iter: Number of iterations
            mix: Mixing parameter for field update

        Returns:
            Dictionary with converged fields and densities
        """
        # Initial guess: random perturbation
        w_minus = 0.1 * torch.randn(self.Nx, self.Nx, self.Nx)
        w_plus = torch.zeros_like(w_minus)

        for iteration in range(n_iter):
            # Fields acting on each species
            w_A = w_plus + w_minus
            w_B = w_plus - w_minus

            # Compute densities
            phi_A, phi_B, Q = self.compute_densities(w_A, w_B)

            # Compute residuals
            incomp = phi_A + phi_B - 1.0
            exchange = phi_A - phi_B - 2 * w_minus / self.chi_N

            error = torch.sqrt((incomp**2).mean() + (exchange**2).mean())

            if iteration % 10 == 0:
```

```
                print(f"Iter {iteration}: error = {error:.6f}")

            if error < 1e-6:
                break

            # Update fields (simple mixing)
            w_plus = w_plus + mix * incomp * self.chi_N
            w_minus = w_minus + mix * exchange * self.chi_N / 2

        return {
            'phi_A': phi_A,
            'phi_B': phi_B,
            'w_A': w_A,
            'w_B': w_B,
            'Q': Q,
            'free_energy': self.compute_free_energy(phi_A, phi_B, w_A, w_B,
    Q)
        }

    def compute_free_energy(self, phi_A, phi_B, w_A, w_B, Q):
        """Compute Helmholtz free energy."""
        V = self.Lx**3
        n = V  # Number density = 1 in Rg units

        # Free energy per chain
        F = -torch.log(Q)
        F += (1/V) * torch.sum(self.chi_N * phi_A * phi_B)
        F -= (1/V) * torch.sum(w_A * phi_A + w_B * phi_B)

        return F.item()
```

12.5 Phase Behavior of Block Copolymers

12.5.1 Equilibrium Morphologies

For diblock copolymers, SCFT predicts:

Morphology	f Range	Symmetry
Spheres (BCC)	$0 < f < 0.12$	$Im\bar{3}m$
Cylinders (HEX)	$0.12 < f < 0.28$	p6mm
Gyroid	$0.28 < f < 0.34$	$Ia\bar{3}d$
Lamellae	$0.34 < f < 0.66$	Lamellar
Gyroid	$0.66 < f < 0.72$	$Ia\bar{3}d$
Cylinders	$0.72 < f < 0.88$	p6mm
Spheres	$0.88 < f < 1$	$Im\bar{3}m$

12.5.2 Order-Disorder Transition

The critical point for symmetric diblocks ($f = 0.5$):

$$(\chi N)_{\text{ODT}} \approx 10.5 \tag{12.13}$$

For asymmetric diblocks, the ODT shifts to higher χN.

12.6 Extensions

12.6.1 Multiblock Copolymers

For ABC triblocks, additional fields and propagators needed.

12.6.2 Polymer Blends

Two chain types with different propagator equations.

12.6.3 Polymer Solutions

Include solvent as explicit small molecules or implicit field.

12.6.4 Confinement

Boundary conditions on fields and propagators.

12.7 Connecting SCFT to MD

12.7.1 SCFT as Coarse-Grained Limit

In the limit of long chains ($N \to \infty$):

- Chain fluctuations suppressed
- Mean-field becomes exact
- SCFT = free energy functional theory

12.7.2 Parameter Mapping

Parameter	MD	SCFT
Chain length	N beads	N (statistical segments)
Interaction	ε, σ	χN
Temperature	T	Implicit in χ
Density	$\rho = N/V$	$\rho_0 = 1/v_0$ (reference)

12.7.3 χ Parameter from MD

The Flory-Huggins χ can be extracted from MD:

$$\chi = \frac{z}{2k_{\mathrm{B}}T}\left[\varepsilon_{AB} - \frac{1}{2}(\varepsilon_{AA} + \varepsilon_{BB})\right] \tag{12.14}$$

Or from coordination number analysis:

$$\chi = \frac{\langle n_{AB}\rangle}{N_A} - \frac{\langle n_{AA}\rangle}{N_A}\frac{\phi_B}{\phi_A} \tag{12.15}$$

12.8 Summary

Concept	Key Point
Field transformation	Particle-particle $\rightarrow$ particle-field
Propagator equation	Modified diffusion equation
Saddle-point approximation	Mean-field = self-consistent fields
Pseudospectral method	FFT-based efficient solver
Phase diagram	Morphology vs f and χN

12.9 Exercises

1. Derive the modified diffusion equation for the chain propagator.
2. Implement a 1D SCFT solver for a lamellar phase and compute the domain spacing as a function of χN.
3. Show that for a homopolymer (no χ), SCFT reduces to the Gaussian chain model.
4. Calculate the free energy difference between lamellar and cylindrical phases for $f = 0.3$, $\chi N = 20$.

12.10 Further Reading

- Fredrickson, G.H. *The Equilibrium Theory of Inhomogeneous Polymers*, Oxford (2006)
- Matsen, M.W. "The Standard Gaussian Model for Block Copolymer Melts," J. Phys.: Condens. Matter **14**, R21 (2002)
- Schmid, F. "Self-Consistent Field Approach for Cross-Linked Copolymer Materials," Phys. Rev. Lett. **111**, 028303 (2013)

Chapter 13

Bridging Scales: MD to Continuum

13.1 Introduction to Multiscale Modeling

Polymer systems span an enormous range of length and time scales, from bond vibrations (femtoseconds, Angstroms) to macroscopic processing (seconds, meters). No single simulation method can span this range; instead, we must bridge scales by connecting models at different levels of description.

Definition 13.1 (Multiscale Modeling)**.** *Multiscale modeling connects simulations at different resolutions, passing information between scales to enable predictions of macroscopic behavior from molecular-level understanding.*

13.2 The Hierarchy of Scales

Method	Length Scale	Time Scale	Information
Quantum (DFT)	1–10 Å	fs	Force field parameters
Atomistic MD	1–100 nm	ns–μs	Local structure, dynamics
Coarse-grained MD	10–1000 nm	μs–ms	Chain conformations
Field theory (SCFT)	100 nm–μm	Equilibrium	Morphology, phase behavior
Mesoscale (DPD)	100 nm–10 μm	μs–ms	Flow, mixing
Continuum (FEM)	μm–m	ms–s	Mechanical response

13.3 Coarse-Graining Strategies

13.3.1 Structural Coarse-Graining

Map atomistic structure to CG beads that reproduce target distributions:

```
def iterative_boltzmann_inversion(r_target, g_target, U_initial,
                                  n_iterations=20, alpha=0.1):
    """
    Iterative Boltzmann Inversion to derive CG potential.

    U_new(r) = U_old(r) + alpha * kT * ln(g_current(r)/g_target(r))
    """
    import numpy as np

    U = U_initial.copy()
    kT = 1.0  # In reduced units
```

```
    for iteration in range(n_iterations):
        # Run CG simulation with current potential
        g_current = run_cg_simulation_get_rdf(U)

        # Update potential
        ratio = g_current / g_target
        ratio[ratio <= 0] = 1e-10  # Avoid log(0)

        delta_U = alpha * kT * np.log(ratio)
        U = U + delta_U

        # Check convergence
        error = np.sqrt(np.mean((g_current - g_target)**2))
        print(f"Iteration {iteration}: RDF error = {error:.4f}")

        if error < 0.01:
            break

    return U
```

13.3.2 Force Matching

Match CG forces to atomistic reference:

$$\min_{\{U_{CG}\}} \sum_{n=1}^{N_{\text{config}}} \sum_{I=1}^{N_{CG}} \left| \mathbf{F}_I^{\text{CG}} - \mathbf{F}_I^{\text{ref}} \right|^2 \tag{13.1}$$

```
def force_matching(atomistic_trajectory, mapping, basis_functions):
    """
    Derive CG potential via force matching.

    Args:
        atomistic_trajectory: AA trajectory with forces
        mapping: Atom-to-bead mapping scheme
        basis_functions: Basis for CG potential expansion

    Returns:
        Coefficients for CG potential
    """
    import numpy as np
    from scipy.optimize import least_squares

    # Map atomistic positions/forces to CG
    cg_positions = []
    cg_forces_ref = []

    for frame in atomistic_trajectory:
        pos_cg, forces_cg = map_aa_to_cg(frame, mapping)
        cg_positions.append(pos_cg)
        cg_forces_ref.append(forces_cg)

    cg_positions = np.array(cg_positions)
    cg_forces_ref = np.array(cg_forces_ref)

    # Setup linear system: F = -sum_k c_k * grad(phi_k)
    def residual(coeffs):
```

```
        forces_model = compute_cg_forces(cg_positions, basis_functions,
    coeffs)
        return (forces_model - cg_forces_ref).flatten()

    # Initial guess
    n_basis = len(basis_functions)
    c0 = np.zeros(n_basis)

    result = least_squares(residual, c0)

    return result.x
```

13.3.3 Relative Entropy Minimization

Minimize the Kullback-Leibler divergence between AA and CG distributions:

$$S_{\text{rel}} = \int P_{AA}(\mathbf{R}) \ln \frac{P_{AA}(\mathbf{R})}{P_{CG}(\mathbf{R};\theta)} \, \mathrm{d}\mathbf{R} \tag{13.2}$$

```
def relative_entropy_optimization(aa_trajectory, cg_potential_func,
                                  initial_params):
    """
    Optimize CG potential by minimizing relative entropy.
    """
    import numpy as np
    from scipy.optimize import minimize

    def objective(params):
        # Compute relative entropy
        S_rel = 0
        for frame in aa_trajectory:
            # Map to CG
            cg_config = map_to_cg(frame)

            # CG energy
            U_cg = cg_potential_func(cg_config, params)

            # Add to relative entropy estimate
            # S_rel = <U_CG>_AA - F_CG
            S_rel += U_cg

        # Need to subtract free energy of CG system
        F_cg = estimate_free_energy_cg(params)

        return S_rel / len(aa_trajectory) - F_cg

    result = minimize(objective, initial_params, method='L-BFGS-B')

    return result.x
```

13.4 Backmapping: CG to Atomistic

13.4.1 Random Placement with Relaxation

```
def backmap_cg_to_aa(cg_positions, topology, n_atoms_per_bead):
    """
    Backmap CG configuration to atomistic resolution.
    """
    import numpy as np

    n_beads = len(cg_positions)
    n_atoms = n_beads * n_atoms_per_bead

    aa_positions = np.zeros((n_atoms, 3))

    for bead_idx in range(n_beads):
        bead_pos = cg_positions[bead_idx]

        # Place atoms randomly around bead center
        for atom_idx in range(n_atoms_per_bead):
            global_idx = bead_idx * n_atoms_per_bead + atom_idx

            # Random displacement within bead radius
            r = np.random.normal(0, 1.0, 3)  # 1 Angstrom spread
            aa_positions[global_idx] = bead_pos + r

    # Local relaxation to remove overlaps
    aa_positions = energy_minimize_local(aa_positions, topology)

    return aa_positions


def geometric_backmapping(cg_positions, fragment_library, topology):
    """
    Backmapping using pre-equilibrated fragment library.
    """
    import numpy as np

    aa_positions = []

    for bead_idx, bead_type in enumerate(topology.bead_types):
        # Select random fragment from library
        fragment = select_fragment(fragment_library, bead_type)

        # Orient fragment to match local CG orientation
        if bead_idx > 0 and bead_idx < len(cg_positions) - 1:
            # Local tangent vector
            tangent = cg_positions[bead_idx + 1] - cg_positions[bead_idx -
   1]
            tangent /= np.linalg.norm(tangent)

            fragment = orient_fragment(fragment, tangent)

        # Translate to bead position
        fragment = fragment + cg_positions[bead_idx]
        aa_positions.extend(fragment)

    return np.array(aa_positions)
```

13.5 Connecting to Continuum Models

13.5.1 Extracting Constitutive Relations

Continuum mechanics uses constitutive relations relating stress to strain/strain-rate:

```
def extract_constitutive_relation(md_trajectories, deformation_type='shear'
    ):
    """
    Extract stress-strain or stress-strain-rate relation from MD.
    """
    import numpy as np
    from scipy.optimize import curve_fit

    if deformation_type == 'shear':
        # Collect shear stress vs shear rate data
        shear_rates = []
        shear_stresses = []

        for traj in md_trajectories:
            rate = traj.metadata['shear_rate']
            stress = np.mean([frame.stress[0,1] for frame in traj])
            shear_rates.append(rate)
            shear_stresses.append(stress)

        shear_rates = np.array(shear_rates)
        shear_stresses = np.array(shear_stresses)

        # Fit power-law model: sigma = K * gamma_dot^n
        def power_law(gamma_dot, K, n):
            return K * gamma_dot**n

        popt, _ = curve_fit(power_law, shear_rates, shear_stresses)
        K, n = popt

        return {'type': 'power_law', 'K': K, 'n': n}

    elif deformation_type == 'extensional':
        # Extensional viscosity
        pass

    return None
```

13.5.2 Viscoelastic Models

Connect MD relaxation data to continuum viscoelastic models:

Maxwell model:

$$\sigma + \lambda \dot{\sigma} = \eta \dot{\gamma} \tag{13.3}$$

Generalized Maxwell (Prony series):

$$G(t) = G_\infty + \sum_{i=1}^{N} G_i \exp(-t/\tau_i) \tag{13.4}$$

```
def fit_prony_series(time, G_t, n_modes=5):
    """
    Fit Prony series to relaxation modulus from MD.
```

```
    """
    import numpy as np
    from scipy.optimize import minimize

    def prony_model(t, G_inf, G_modes, tau_modes):
        G = G_inf * np.ones_like(t)
        for Gi, taui in zip(G_modes, tau_modes):
            G += Gi * np.exp(-t / taui)
        return G

    def objective(params):
        G_inf = params[0]
        G_modes = params[1:n_modes+1]
        tau_modes = params[n_modes+1:]

        G_fit = prony_model(time, G_inf, G_modes, tau_modes)
        return np.sum((G_fit - G_t)**2)

    # Initial guess: log-spaced relaxation times
    tau_init = np.logspace(np.log10(time[1]), np.log10(time[-1]/2), n_modes
    )
    G_init = np.ones(n_modes) * G_t[0] / n_modes

    x0 = np.concatenate([[G_t[-1]], G_init, tau_init])

    # Bounds
    bounds = [(0, None)] * (2*n_modes + 1)

    result = minimize(objective, x0, bounds=bounds, method='L-BFGS-B')

    G_inf = result.x[0]
    G_modes = result.x[1:n_modes+1]
    tau_modes = result.x[n_modes+1:]

    return G_inf, G_modes, tau_modes
```

13.5.3 Material Parameters for FEM

```
def generate_fem_material_card(md_results, material_model='neo_hookean'):
    """
    Generate material card for FEM software from MD results.
    """
    if material_model == 'neo_hookean':
        # G from MD rubber elasticity
        G = md_results['shear_modulus']
        K = md_results['bulk_modulus']

        material_card = f"""
*MATERIAL, NAME=POLYMER_MD
*HYPERELASTIC, NEO HOOKE
{G/2:.6e}, {K:.6e}
"""

    elif material_model == 'viscoelastic':
        G_inf, G_modes, tau_modes = md_results['prony_series']

        material_card = f"""
```

```
*MATERIAL, NAME=POLYMER_MD
*ELASTIC
{md_results['E']:.6e}, {md_results['nu']:.4f}
*VISCOELASTIC, TIME=PRONY
"""
        for G_i, tau_i in zip(G_modes, tau_modes):
            g_i = G_i / (G_inf + sum(G_modes))  # Normalized
            material_card += f"{g_i:.6e}, 0.0, {tau_i:.6e}\n"

    return material_card
```

13.6 Tube Model Parameters from MD

13.6.1 Primitive Path Analysis

Extract entanglement parameters for tube-based continuum models:

```
def primitive_path_analysis(positions, chain_indices, box_size):
    """
    Extract primitive path by minimizing contour length
    while preserving topology.

    Returns entanglement length Ne and tube diameter a.
    """
    import numpy as np

    # Implementation of Z1 or CReTA algorithm
    # Simplified version:

    n_chains = len(chain_indices)
    entanglement_lengths = []

    for chain_idx in range(n_chains):
        chain_pos = positions[chain_indices[chain_idx]]

        # Contract chain while maintaining topology
        pp_pos = contract_chain_preserve_topology(chain_pos, positions,
                                                  chain_indices)

        # Primitive path length
        L_pp = compute_contour_length(pp_pos)

        # Original contour length
        L_0 = compute_contour_length(chain_pos)

        # Number of entanglement strands
        Z = L_0**2 / (L_pp * L_0)

        N = len(chain_indices[chain_idx])
        Ne = N / Z

        entanglement_lengths.append(Ne)

    Ne_avg = np.mean(entanglement_lengths)

    # Tube diameter from Ne
    b = 1.54  # Bond length in Angstrom
```

```
    Cinf = 7.4  # For PE
    a = np.sqrt(Ne_avg * Cinf) * b

    return Ne_avg, a


def compute_tube_diameter_from_msd(time, msd, N, b=1.54):
    """
    Extract tube diameter from MSD plateau.

    In tube regime: MSD ~ a^2 (localization)
    """
    import numpy as np

    # Find plateau region
    d_msd = np.gradient(np.log(msd), np.log(time))

    # Plateau where d(log MSD)/d(log t) is minimum
    plateau_idx = np.argmin(np.abs(d_msd - 0.25))  # Rouse in tube: t^0.25

    # Tube diameter
    a_sq = msd[plateau_idx]
    a = np.sqrt(a_sq)

    return a
```

13.6.2 Entanglement Molecular Weight

```
def compute_Me_from_plateau_modulus(G_N0, density, temperature):
    """
    Compute entanglement molecular weight from plateau modulus.

    Me = rho * R * T / G_N0
    """
    R = 8.314  # J/(mol*K)

    Me = density * R * temperature / G_N0

    return Me
```

13.7 Workflow: MD to Continuum

```
def complete_multiscale_workflow(atomistic_structure):
    """
    Complete workflow from atomistic to continuum.
    """
    results = {}

    # Step 1: Atomistic equilibration and property calculation
    aa_system = setup_atomistic_system(atomistic_structure)
    aa_system.equilibrate(time=10e-9)  # 10 ns

    results['aa_Rg'] = compute_Rg(aa_system)
    results['aa_density'] = aa_system.density
```

```
    # Step 2: Derive CG potential
    cg_potential = derive_cg_potential(aa_system.trajectory,
                                       method='force_matching')

    # Step 3: CG simulation
    cg_system = create_cg_system(aa_system, cg_potential)
    cg_system.equilibrate(time=1e-6)  # 1 us

    results['cg_Rg'] = compute_Rg(cg_system)
    results['relaxation_times'] = compute_relaxation_times(cg_system)

    # Step 4: Extract entanglement parameters
    Ne, tube_diameter = primitive_path_analysis(cg_system)
    results['Ne'] = Ne
    results['tube_diameter'] = tube_diameter

    # Step 5: Mechanical properties
    G_t = compute_relaxation_modulus(cg_system)
    G_inf, G_modes, tau_modes = fit_prony_series(G_t)

    results['G_inf'] = G_inf
    results['prony_series'] = (G_modes, tau_modes)

    # Step 6: Generate continuum model
    fem_card = generate_fem_material_card(results)
    results['fem_material_card'] = fem_card

    return results
```

13.8 Validation of Scale Bridging

13.8.1 Consistency Checks

```
def validate_coarse_graining(aa_trajectory, cg_trajectory, mapping):
    """
    Validate CG model reproduces AA target properties.
    """
    metrics = {}

    # 1. Structural comparison
    r_aa, g_aa = compute_rdf(aa_trajectory)
    r_cg, g_cg = compute_rdf(cg_trajectory)

    # Map AA RDF to CG resolution
    g_aa_mapped = map_rdf_to_cg(r_aa, g_aa, mapping)

    metrics['rdf_error'] = np.sqrt(np.mean((g_aa_mapped - g_cg)**2))

    # 2. Chain dimensions
    Rg_aa = compute_Rg(aa_trajectory)
    Rg_cg = compute_Rg(cg_trajectory)

    metrics['Rg_error'] = abs(Rg_aa - Rg_cg) / Rg_aa

    # 3. Pressure/density
    P_aa = np.mean([f.pressure for f in aa_trajectory])
```

```
    P_cg = np.mean([f.pressure for f in cg_trajectory])

    metrics['pressure_error'] = abs(P_aa - P_cg) / abs(P_aa)

    # 4. Dynamic consistency (diffusion timescale)
    D_aa = compute_diffusion(aa_trajectory)
    D_cg = compute_diffusion(cg_trajectory)

    metrics['diffusion_ratio'] = D_cg / D_aa  # Should be >1 due to speedup

    return metrics
```

13.9 Summary

Scale Transition	Key Methods
Atomistic → CG	IBI, force matching, relative entropy
CG → Atomistic	Geometric backmapping, relaxation
MD → Continuum	Constitutive relations, Prony series, tube parameters
Validation	RDF matching, chain dimensions, dynamics

13.10 Exercises

1. Derive a CG potential for polyethylene using IBI and validate against atomistic $g(r)$.
2. Extract entanglement molecular weight from the plateau modulus of a polymer melt.
3. Fit a Prony series to MD relaxation data and compare with experimental DMA.
4. Implement a backmapping procedure and validate by comparing backmapped structure to original.
5. Generate FEM material parameters from MD and validate with a simple deformation simulation.

13.11 Further Reading

- Padding, J.T. & Briels, W.J. "Systematic coarse-graining of the dynamics of entangled polymer melts"
- Noid, W.G. "Perspective: Coarse-grained models for biomolecular systems"
- Kröger, M. "Shortest multiple disconnected path for the analysis of entanglements"

Chapter 14

Machine Learning for Polymer Simulations

14.1 Introduction to ML in Polymer Science

Machine learning (ML) is transforming computational polymer science by enabling faster simulations, more accurate predictions, and discovery of new materials. This chapter covers the key ML approaches relevant to polymer MD simulations.

Definition 14.1 (Machine Learning Potential)**.** *A machine learning potential (MLP) is a computational model that predicts interatomic energies and forces from atomic configurations, trained on reference quantum mechanical or higher-level calculations.*

14.2 Machine Learning Potentials

14.2.1 Why ML Potentials for Polymers?

- **Speed**: 100–1000× faster than DFT while approaching DFT accuracy
- **Accuracy**: Better than classical force fields for reactive systems
- **Transferability**: Can capture complex many-body interactions
- **Automation**: Reduce human effort in force field development

14.2.2 Neural Network Potentials

Behler-Parrinello architecture:

$$E = \sum_i E_i(\{G_i\}) \tag{14.1}$$

where E_i is the atomic energy from a neural network and $\{G_i\}$ are symmetry functions describing the local environment.

Symmetry functions (radial):

$$G_i^{\text{rad}} = \sum_j e^{-\eta(R_{ij}-R_s)^2} f_c(R_{ij}) \tag{14.2}$$

Symmetry functions (angular):

$$G_i^{\text{ang}} = 2^{1-\zeta} \sum_{j,k\neq i} (1+\lambda\cos\theta_{ijk})^{\zeta} e^{-\eta(R_{ij}^2+R_{ik}^2+R_{jk}^2)} f_c(R_{ij}) f_c(R_{ik}) f_c(R_{jk}) \tag{14.3}$$

```
import torch
import torch.nn as nn

class AtomicNeuralNetwork(nn.Module):
    """
    Atomic neural network for energy prediction.
    """
    def __init__(self, n_symmetry_functions, hidden_layers=[64, 64, 64]):
        super().__init__()

        layers = []
        input_size = n_symmetry_functions

        for hidden_size in hidden_layers:
            layers.append(nn.Linear(input_size, hidden_size))
            layers.append(nn.Tanh())
            input_size = hidden_size

        layers.append(nn.Linear(input_size, 1))  # Output: atomic energy

        self.network = nn.Sequential(*layers)

    def forward(self, symmetry_functions):
        """
        Args:
            symmetry_functions: (n_atoms, n_sf) tensor

        Returns:
            Atomic energies: (n_atoms,) tensor
        """
        atomic_energies = self.network(symmetry_functions).squeeze(-1)
        return atomic_energies


class NeuralNetworkPotential(nn.Module):
    """
    Full neural network potential for molecular systems.
    """
    def __init__(self, element_types, n_symmetry_functions):
        super().__init__()

        # One network per element type
        self.atomic_networks = nn.ModuleDict({
            elem: AtomicNeuralNetwork(n_symmetry_functions)
            for elem in element_types
        })

    def forward(self, positions, elements, neighbor_list):
        # Compute symmetry functions
        G = compute_symmetry_functions(positions, neighbor_list)

        # Compute atomic energies
        total_energy = 0
        for i, elem in enumerate(elements):
            E_i = self.atomic_networks[elem](G[i:i+1])
            total_energy += E_i
```

```
        return total_energy
```

14.2.3 Equivariant Neural Networks

Modern architectures preserve symmetry by construction:

```
# Example using e3nn (equivariant neural networks)
"""
import e3nn
from e3nn import o3
from e3nn.nn import FullyConnectedNet

class EquivariantPolymerPotential(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Irreducible representations
        irreps_input = o3.Irreps("1x0e")  # Scalar (element type)
        irreps_hidden = o3.Irreps("32x0e + 16x1o + 8x2e")
        irreps_output = o3.Irreps("1x0e")  # Energy (scalar)

        self.conv_layers = torch.nn.ModuleList([
            e3nn.nn.Gate(irreps_hidden, [torch.relu], irreps_hidden)
            for _ in range(3)
        ])

    def forward(self, positions, atom_types):
        # Build graph
        edge_index = radius_graph(positions, r=5.0)
        edge_vec = positions[edge_index[1]] - positions[edge_index[0]]

        # Message passing with equivariant operations
        # ...
        return energy
"""
```

14.2.4 Training ML Potentials

```
def train_ml_potential(model, training_data, validation_data,
                       n_epochs=1000, lr=1e-3):
    """
    Train ML potential on reference data.

    Args:
        training_data: List of (positions, energy, forces) tuples
        validation_data: Validation set
    """
    import torch.optim as optim

    optimizer = optim.Adam(model.parameters(), lr=lr)

    # Loss weights
    w_energy = 1.0
    w_forces = 100.0  # Forces are more important for dynamics

    for epoch in range(n_epochs):
```

```
        model.train()
        total_loss = 0

        for positions, E_ref, F_ref in training_data:
            optimizer.zero_grad()

            # Forward pass
            positions.requires_grad_(True)
            E_pred = model(positions)

            # Compute forces from energy gradient
            F_pred = -torch.autograd.grad(E_pred, positions,
                                          create_graph=True)[0]

            # Combined loss
            loss_E = torch.mean((E_pred - E_ref)**2)
            loss_F = torch.mean((F_pred - F_ref)**2)
            loss = w_energy * loss_E + w_forces * loss_F

            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        # Validation
        if epoch % 100 == 0:
            val_error = evaluate_model(model, validation_data)
            print(f"Epoch {epoch}: Train loss = {total_loss:.4f}, "
                  f"Val RMSE = {val_error:.4f}")

    return model
```

14.2.5 Active Learning for Data Generation

```
def active_learning_loop(initial_model, md_engine, n_iterations=10):
    """
    Active learning to improve ML potential coverage.
    """
    model = initial_model
    training_data = []

    for iteration in range(n_iterations):
        # Run MD with current model
        trajectory = md_engine.run(model, steps=10000)

        # Identify uncertain configurations
        uncertain_configs = identify_high_uncertainty(model, trajectory)

        # Compute reference (DFT) for uncertain configs
        new_data = []
        for config in uncertain_configs[:10]:  # Limit expensive
   calculations
            E_ref, F_ref = compute_dft_reference(config)
            new_data.append((config, E_ref, F_ref))

        training_data.extend(new_data)
```

```
        # Retrain model
        model = train_ml_potential(model, training_data)

        print(f"Iteration {iteration}: Added {len(new_data)} configurations
    ")

    return model


def identify_high_uncertainty(model, trajectory, method='committee'):
    """
    Identify configurations with high model uncertainty.
    """
    if method == 'committee':
        # Use ensemble of models
        predictions = []
        for member in model.ensemble:
            pred = [member(config) for config in trajectory]
            predictions.append(pred)

        # Uncertainty = standard deviation across ensemble
        uncertainty = np.std(predictions, axis=0)

        # Select high-uncertainty configurations
        high_unc_idx = np.argsort(uncertainty)[-20:]
        return [trajectory[i] for i in high_unc_idx]
```

14.3 Property Prediction with ML

14.3.1 Structure-Property Models

Predict polymer properties from molecular descriptors:

```
def create_polymer_descriptors(smiles_or_structure):
    """
    Create feature vector for polymer property prediction.
    """
    from rdkit import Chem
    from rdkit.Chem import Descriptors, AllChem

    # Monomer descriptors
    mol = Chem.MolFromSmiles(smiles_or_structure)

    descriptors = {
        'MW': Descriptors.MolWt(mol),
        'logP': Descriptors.MolLogP(mol),
        'TPSA': Descriptors.TPSA(mol),
        'n_rotatable': Descriptors.NumRotatableBonds(mol),
        'n_HBD': Descriptors.NumHDonors(mol),
        'n_HBA': Descriptors.NumHAcceptors(mol),
        'n_aromatic_rings': Descriptors.NumAromaticRings(mol),
    }

    # Morgan fingerprint
    fp = AllChem.GetMorganFingerprintAsBitVect(mol, 2, nBits=2048)
    descriptors['fingerprint'] = list(fp)
```

```
    return descriptors


def train_property_predictor(polymer_data, target_property='Tg'):
    """
    Train ML model to predict polymer property.
    """
    from sklearn.ensemble import GradientBoostingRegressor
    from sklearn.model_selection import cross_val_score
    import numpy as np

    # Prepare features
    X = []
    y = []

    for polymer in polymer_data:
        desc = create_polymer_descriptors(polymer['smiles'])
        features = [desc['MW'], desc['logP'], desc['TPSA'],
                    desc['n_rotatable']] + desc['fingerprint']
        X.append(features)
        y.append(polymer[target_property])

    X = np.array(X)
    y = np.array(y)

    # Train model
    model = GradientBoostingRegressor(n_estimators=100, max_depth=5)
    model.fit(X, y)

    # Cross-validation
    scores = cross_val_score(model, X, y, cv=5,
                             scoring='neg_mean_absolute_error')
    print(f"CV MAE: {-np.mean(scores):.2f} +/- {np.std(scores):.2f}")

    return model
```

14.3.2 Graph Neural Networks for Polymers

```
import torch
import torch.nn as nn
from torch_geometric.nn import GCNConv, global_mean_pool

class PolymerGNN(nn.Module):
    """
    Graph neural network for polymer property prediction.
    """
    def __init__(self, node_features, hidden_dim=64, n_layers=3):
        super().__init__()

        self.convs = nn.ModuleList()
        self.convs.append(GCNConv(node_features, hidden_dim))

        for _ in range(n_layers - 1):
            self.convs.append(GCNConv(hidden_dim, hidden_dim))

        self.fc = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
```

```
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x, edge_index, batch):
        """
        Args:
            x: Node features (n_nodes, node_features)
            edge_index: Graph connectivity (2, n_edges)
            batch: Batch assignment (n_nodes,)
        """
        for conv in self.convs:
            x = conv(x, edge_index)
            x = torch.relu(x)

        # Global pooling
        x = global_mean_pool(x, batch)

        # Predict property
        return self.fc(x)


def polymer_to_graph(smiles):
    """
    Convert polymer SMILES to graph representation.
    """
    from rdkit import Chem
    import torch

    mol = Chem.MolFromSmiles(smiles)

    # Node features: atomic properties
    node_features = []
    for atom in mol.GetAtoms():
        features = [
            atom.GetAtomicNum(),
            atom.GetDegree(),
            atom.GetFormalCharge(),
            atom.GetNumRadicalElectrons(),
            atom.GetHybridization().real,
            atom.GetIsAromatic(),
        ]
        node_features.append(features)

    x = torch.tensor(node_features, dtype=torch.float)

    # Edge index
    edges = []
    for bond in mol.GetBonds():
        i = bond.GetBeginAtomIdx()
        j = bond.GetEndAtomIdx()
        edges.append([i, j])
        edges.append([j, i])  # Undirected

    edge_index = torch.tensor(edges, dtype=torch.long).T

    return x, edge_index
```

14.4 Generative Models for Polymer Design

14.4.1 Variational Autoencoder (VAE)

```
class PolymerVAE(nn.Module):
    """
    Variational autoencoder for polymer generation.
    """
    def __init__(self, vocab_size, embedding_dim=64, latent_dim=128):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # Encoder
        self.encoder = nn.LSTM(embedding_dim, 256, batch_first=True,
                               bidirectional=True)
        self.fc_mu = nn.Linear(512, latent_dim)
        self.fc_var = nn.Linear(512, latent_dim)

        # Decoder
        self.fc_decode = nn.Linear(latent_dim, 256)
        self.decoder = nn.LSTM(256, 256, batch_first=True)
        self.output = nn.Linear(256, vocab_size)

    def encode(self, x):
        embedded = self.embedding(x)
        _, (h, _) = self.encoder(embedded)
        h = torch.cat([h[-2], h[-1]], dim=1)

        mu = self.fc_mu(h)
        log_var = self.fc_var(h)

        return mu, log_var

    def reparameterize(self, mu, log_var):
        std = torch.exp(0.5 * log_var)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z, max_length=100):
        h = self.fc_decode(z).unsqueeze(0)
        c = torch.zeros_like(h)

        outputs = []
        input_token = torch.zeros(z.size(0), 1, 256).to(z.device)

        for _ in range(max_length):
            output, (h, c) = self.decoder(input_token, (h, c))
            logits = self.output(output)
            outputs.append(logits)
            input_token = output

        return torch.cat(outputs, dim=1)

    def generate(self, n_samples=10):
        z = torch.randn(n_samples, self.latent_dim)
        return self.decode(z)
```

14.4.2 Reinforcement Learning for Optimization

```
class PolymerRLAgent:
    """
    RL agent for polymer optimization.
    """
    def __init__(self, policy_network, property_predictor,
                 target_property='Tg', target_value=400):
        self.policy = policy_network
        self.predictor = property_predictor
        self.target = target_value

    def compute_reward(self, polymer_smiles):
        """
        Reward based on predicted property and validity.
        """
        # Check validity
        mol = Chem.MolFromSmiles(polymer_smiles)
        if mol is None:
            return -1.0  # Invalid

        # Predict property
        descriptors = create_polymer_descriptors(polymer_smiles)
        predicted_value = self.predictor.predict([descriptors])[0]

        # Reward: negative distance from target
        reward = -abs(predicted_value - self.target) / self.target

        # Bonus for synthesizability
        reward += 0.1 * compute_sa_score(mol)

        return reward

    def train_episode(self, optimizer):
        """
        Run one episode of RL training.
        """
        # Generate polymer
        polymer = self.policy.generate(temperature=1.0)

        # Compute reward
        reward = self.compute_reward(polymer)

        # Policy gradient update
        loss = -reward * self.policy.log_prob(polymer)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        return polymer, reward
```

14.5 Accelerating MD with ML

14.5.1 Collective Variable Learning

Learn collective variables for enhanced sampling:

```
class CVAutoencoder(nn.Module):
    """
    Autoencoder to learn collective variables.
    """
    def __init__(self, input_dim, cv_dim=2):
        super().__init__()

        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, cv_dim)
        )

        self.decoder = nn.Sequential(
            nn.Linear(cv_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, input_dim)
        )

    def forward(self, x):
        cv = self.encoder(x)
        reconstructed = self.decoder(cv)
        return cv, reconstructed


def train_cv_autoencoder(trajectory, n_cvs=2):
    """
    Train autoencoder to find collective variables.
    """
    # Prepare data: flatten positions
    X = np.array([frame.positions.flatten() for frame in trajectory])
    X = torch.tensor(X, dtype=torch.float32)

    model = CVAutoencoder(X.shape[1], cv_dim=n_cvs)
    optimizer = torch.optim.Adam(model.parameters())

    for epoch in range(1000):
        cv, reconstructed = model(X)
        loss = nn.MSELoss()(reconstructed, X)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    return model
```

14.5.2 Surrogate Models for Property Calculation

```
class PropertySurrogate(nn.Module):
    """
    Surrogate model to predict expensive properties from configurations.
    """
```

```
    def __init__(self, input_features, output_dim=1):
        super().__init__()

        self.network = nn.Sequential(
            nn.Linear(input_features, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, output_dim)
        )

    def forward(self, x):
        return self.network(x)


def train_rg_surrogate(trajectories):
    """
    Train surrogate to predict Rg from configuration.
    """
    X = []  # Features from configuration
    y = []  # Rg values

    for traj in trajectories:
        for frame in traj:
            features = compute_configuration_features(frame)
            rg = compute_Rg(frame)

            X.append(features)
            y.append(rg)

    X = torch.tensor(X, dtype=torch.float32)
    y = torch.tensor(y, dtype=torch.float32).unsqueeze(1)

    model = PropertySurrogate(X.shape[1])
    optimizer = torch.optim.Adam(model.parameters())

    for epoch in range(500):
        pred = model(X)
        loss = nn.MSELoss()(pred, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    return model
```

14.6 Summary

ML Application	Use Case
ML potentials	Replace force fields with QM accuracy at classical speed
Property prediction	QSPR for T_g, modulus, permeability
Generative models	Design polymers with target properties
CV learning	Enhanced sampling, reaction coordinates
Surrogates	Fast approximation of expensive calculations

14.7 Exercises

1. Train a neural network potential for polyethylene using symmetry functions.
2. Build a GNN to predict glass transition temperature from polymer structure.
3. Implement a VAE for polymer SMILES and generate novel structures.
4. Use active learning to improve an ML potential's coverage of configuration space.
5. Train an autoencoder to find collective variables for polymer crystallization.

14.8 Further Reading

- Behler, J. "Perspective: Machine learning potentials for atomistic simulations"
- Batra, R. et al. "Polymers for Extreme Conditions Designed Using Machine Learning"
- Noé, F. et al. "Machine Learning for Molecular Simulation"
- Gómez-Bombarelli, R. et al. "Design of efficient molecular organic light-emitting diodes by a high-throughput virtual screening approach"

Quick Reference

Key Polymer Equations

Property	Equation	Description
End-to-end distance	$\langle R_{\text{ee}}^2 \rangle = C_\infty n b^2$	Characteristic ratio scaling
Radius of gyration	$\langle R_{\text{g}}^2 \rangle = \langle R_{\text{ee}}^2 \rangle / 6$	For Gaussian chains
Kuhn length	$l_{\text{K}} = C_\infty b$	Effective segment length
Persistence length	$l_{\text{p}} = -b / \ln\langle \cos\theta \rangle$	From bond correlations
Flory exponent	$R_{\text{g}} \sim N^\nu$	$\nu = 0.588$ (good solvent)
Rouse time	$\tau_R = \zeta N^2 b^2 / (3\pi^2 k_{\text{B}} T)$	Unentangled relaxation

Key Force Field Parameters

Potential	Formula	Typical Parameters	Use
FENE	$-\frac{1}{2} k R_0^2 \ln\left(1 - \frac{r^2}{R_0^2}\right)$	$k = 30\varepsilon/\sigma^2$, $R_0 = 1.5\sigma$	CG bonds
WCA	$4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] + \varepsilon$	$r_c = 2^{1/6}\sigma$	Excluded volume
Cosine bend	$\kappa\varepsilon(1 + \cos\theta)$	$\kappa \approx 2.16$ (PE)	Chain stiffness
Harmonic bond	$\frac{1}{2} k (r - r_0)^2$	$k \sim 300$ kcal/mol/Å^2	Atomistic

Characteristic Ratios for Common Polymers

Polymer	C_∞	κ (KG model)	l_{K} (Å)	T_g (K)
Polyethylene (PE)	6.7–7.4	2.16	14.5	195
Polypropylene (iPP)	5.5–5.9	1.5	11.2	270
Polystyrene (aPS)	9.5–10.5	0.94	18.0	373
PDMS	6.2–6.5	0	13.0	150
PMMA	8.2–9.0	—	17.0	378

Python Quick Start

```
# Initialize a Kremer-Grest polymer simulation
import torch
from icme_polymer.coarse_grained import CoarseGrainedMD
from icme_polymer.core import POLYOLEFIN_LIBRARY, PolymerType

# Get parameters for HDPE (kappa=2.16 for C_inf ~ 7.4)
params = POLYOLEFIN_LIBRARY[PolymerType.HDPE]
```

```
# Create MD engine
md = CoarseGrainedMD(positions, bonds, angles, params, device='cuda')

# Run equilibration + production
results = md.run_equilibration(
    n_equil=10000,
    n_prod=50000,
    temperature=450.0,  # Above Tg
    dt=0.005
)

# Validate characteristic ratio
validation = md.validate_chain_statistics(
    backbone_length=200,
    target_cn=7.4  # Experimental PE value
)
print(f"C_inf from simulation: {validation['cn_simulation']:.2f}")
print(f"Kuhn length: {validation['kuhn_length_simulation']:.1f} A")
```

Key References

1. Rubinstein, M. & Colby, R.H. (2003). *Polymer Physics*. Oxford University Press.
2. Doi, M. & Edwards, S.F. (1986). *The Theory of Polymer Dynamics*. Oxford University Press.
3. Kremer, K. & Grest, G.S. (1990). J. Chem. Phys. **92**, 5057.
4. Everaers, R. et al. (2020). Macromolecules **53**, 1901–1916.
5. Flory, P.J. (1969). *Statistical Mechanics of Chain Molecules*. Wiley.
6. Allen, M.P. & Tildesley, D.J. (2017). *Computer Simulation of Liquids*. Oxford.
7. Frenkel, D. & Smit, B. (2002). *Understanding Molecular Simulation*. Academic Press.