FrameDiff v2: Unconditional Protein Backbone Generation via IGSO3 Score Matching on SE(3) Frames

Sergio E. Mares

March 2026

Abstract

We present FrameDiff v2, an unconditional generative model for protein backbone structures that operates directly on SE(3) rigid-body frames. The model employs Invariant Point Attention (IPA) within an 8-layer iterative refinement architecture (14.4M parameters) to predict clean frames from noised inputs. Rotational noise follows the Isotropic Gaussian distribution on SO(3) (IGSO3), and the rotation loss uses proper IGSO3 score matching in the tangent space rather than the Frobenius geodesic norm used in v1. Translations follow a variance-preserving SDE with centered Gaussian noise. Auxiliary losses on backbone atom positions and pairwise Cα distances regularize low-noise predictions. Training on CATH 4.3 (10,600 chains, max 256 residues) with cosine learning-rate decay reaches Cα RMSD of 4.45 Å and mean rotation error of 45.5° at epoch 122. We document five numerical stability fixes required for NaN-free training and analyze a training instability at epoch 81 that motivated the switch from constant to cosine LR scheduling.

1 Introduction

Protein backbone generation is a key task in computational structural biology, with applications in *de novo* protein design, scaffold generation for functional-site grafting, and data augmentation for structure prediction. The backbone of a protein can be described as a sequence of rigid-body frames in SE(3): each residue i carries a rotation matrix $R_i \in \mathrm{SO}(3)$ (orienting the peptide plane) and a translation $\mathbf{x}_i \in \mathbb{R}^3$ (the Cα position).

Diffusion models have emerged as powerful generative frameworks for continuous data. However, SO(3) is a compact, non-Euclidean manifold, so standard Gaussian diffusion does not directly apply to rotations. FrameDiff v1 used a Frobenius geodesic loss ($\|\log(R_{\text{pred}}^\top R_{\text{true}})\|^2$) which ignores the proper score-matching structure of the diffusion process on SO(3).

FrameDiff v2 replaces this with the Isotropic Gaussian on SO(3) (IGSO3) distribution, which is the natural analogue of a Gaussian on $\mathbb{R}^n$ but defined on the rotation group. The IGSO3 density admits a closed-form series expansion, enabling exact score computation and proper score-matching loss formulation. This yields a theoretically principled training objective and, in conjunction with architectural improvements (deeper MLPs, IPA transition layers, configurable stop-gradient), leads to improved training dynamics.

2 Model Architecture

The FrameDiff v2 model is an iterative refinement network that takes noised SE(3) frames and a diffusion timestep as input and predicts clean frames. It has no sequence input and no encoder—it is purely unconditional. The architecture consists of $L = 8$ stacked refinement layers, each containing five components described below.

2.1 Invariant Point Attention (IPA)

IPA is the core attention mechanism, adapted from AlphaFold2. It computes attention in both scalar space and 3-D point space, with point queries/keys/values transformed through per-residue SE(3) frames to ensure SE(3) equivariance.

Given node features $\mathbf{h}_i \in \mathbb{R}^{d_{\text{node}}}$, edge features $\mathbf{z}_{ij} \in \mathbb{R}^{d_{\text{edge}}}$, and frames $(R_i, \mathbf{x}_i)$, the attention logits for head a are:

$$\alpha_{ij}^{(a)} = \underbrace{\frac{\mathbf{q}_i^{(a)\top}\mathbf{k}_j^{(a)}}{\sqrt{d_{\text{head}}}}}_{\text{scalar}} - \underbrace{\gamma^{(a)} \sum_{p=1}^{P_q} \|\mathbf{q}_{i,p}^{\text{pt}} - \mathbf{k}_{j,p}^{\text{pt}}\|^2}_{\text{point}} + \underbrace{\mathbf{z}_{ij}^\top \mathbf{w}_{\text{pair}}^{(a)}}_{\text{pair bias}} \tag{1}$$

where $\gamma^{(a)} = \text{softplus}(\hat{\gamma}^{(a)})$ is a learnable per-head weight, and the point queries/keys are transformed to the global frame via $\mathbf{q}_{i,p}^{\text{pt}} = R_i \mathbf{q}_{i,p}^{\text{local}} + \mathbf{x}_i$.

Attention weights are $w_{ij}^{(a)} = \text{softmax}_j(\alpha_{ij}^{(a)})$. The output aggregates scalar values, point value norms (transformed back to the local frame), and pair-weighted values:

$$\mathbf{o}_i = W_{\text{out}} \left[\bigoplus_{a=1}^{H} \left(\sum_j w_{ij}^{(a)} \mathbf{v}_j^{(a)}, \left\| \sum_j w_{ij}^{(a)} \mathbf{v}_{j,p}^{\text{pt,local}} \right\|, \sum_j w_{ij}^{(a)} \mathbf{v}_{ij}^{\text{pair}} \right) \right] \tag{2}$$

The IPA uses $H = 12$ heads, $d_{\text{head}} = 16$, $P_q = 8$ query points, and $P_v = 12$ value points.

2.2 IPA Transition MLP

After IPA and LayerNorm, a 2-layer MLP with ReLU activation is applied with a residual connection (AF2-style):

$$\mathbf{h}_i \leftarrow \mathbf{h}_i + \text{MLP}_{\text{trans}}(\mathbf{h}_i), \quad \text{MLP}_{\text{trans}} : \mathbb{R}^{d_{\text{node}}} \to \mathbb{R}^{d_{\text{node}}} \to \mathbb{R}^{d_{\text{node}}} \tag{3}$$

2.3 Transformer Block with Skip Connections

The node features are concatenated with the initial features $\mathbf{h}_i^{(0)}$ and processed through a Transformer block:

$$\mathbf{h}_i \leftarrow \text{TransformerBlock}\Big([\mathbf{h}_i \,\|\, \mathbf{h}_i^{(0)}]\Big) \tag{4}$$

The block projects $2d_{\text{node}} \to d_{\text{node}}$, applies multi-head self-attention ($n_{\text{heads}} = 4$) with pre-LayerNorm and a 2-layer FFN ($d_{\text{ffn}} = 1024$) with GELU activation, each with residual connections.

2.4 Edge Update

Edge features are updated via a 2-layer MLP (v2 improvement over v1's single linear layer):

$$\mathbf{z}_{ij} \leftarrow \text{LayerNorm}(\mathbf{z}_{ij} + \text{MLP}_{\text{edge}}([\mathbf{h}_i \,\|\, \mathbf{h}_j \,\|\, \mathbf{z}_{ij}])) \tag{5}$$

where $\text{MLP}_{\text{edge}} : \mathbb{R}^{2d_{\text{node}}+d_{\text{edge}}} \to \mathbb{R}^{d_{\text{edge}}} \xrightarrow{\text{ReLU}} \mathbb{R}^{d_{\text{edge}}}$.

2.5 Backbone Update

A 2-layer MLP predicts a small SE(3) update (quaternion + translation) from node features:

$$(\Delta q_i,\, \Delta \mathbf{x}_i) = \text{MLP}_{\text{bb}}(\mathbf{h}_i) \in \mathbb{R}^4 \times \mathbb{R}^3 \tag{6}$$

The quaternion is normalized and converted to a rotation matrix ΔR_i, then composed: $R_i \leftarrow R_i \cdot \Delta R_i$. The resulting matrix is re-orthogonalized via Gram–Schmidt (not SVD, for gradient stability). Translations are updated as $\mathbf{x}_i \leftarrow \mathbf{x}_i + R_i \Delta \mathbf{x}_i$ and re-centered to zero center of mass.

The final layer's weights are initialized near zero with quaternion bias $[1, 0, 0, 0]$ so the initial update is approximately the identity.

2.6 Torsion Head

A 2-layer MLP predicts $(\sin \psi_i, \cos \psi_i)$ from node features, converted to angles via $\psi_i = \text{atan2}(\sin \psi_i, \cos \psi_i)$.

2.7 Self-Conditioning via Distogram

With probability 0.5 during training, the pairwise Cα distance distogram (32 bins, 0.2–2.0 nm) from the previous prediction is projected to 63 dimensions and concatenated with relative position encodings to form edge features. This self-conditioning provides structural context from the model's own prior predictions.

2.8 Stop-Gradient on Intermediate Frames

In v1, gradients through intermediate frame updates were always detached (stop-gradient). In v2, this is configurable via a flag; the default allows gradient flow through all layers, enabling the backbone update MLPs to receive direct supervision signal.

3 IGSO3 Diffusion Process

3.1 IGSO3 Density Function

The Isotropic Gaussian on SO(3) is the heat kernel on the rotation group. For a rotation with angle $\omega \in [0, \pi]$ at diffusion time t, the density is:

$$f(\omega, t) = \sum_{\ell=0}^{L_{\max}} (2\ell + 1) \exp\left(-\frac{\ell(\ell+1)\, t}{2}\right) \frac{\sin\big((\ell + \frac{1}{2})\omega\big)}{\sin(\omega/2)} \tag{7}$$

The series is truncated at $L_{\max} = 1000$, with early termination when $\exp(-\ell(\ell+1)t/2) < 10^{-30}$ for all remaining terms.

At $\omega \to 0$, L'Hôpital's rule gives $\sin((\ell+\frac{1}{2})\omega)/\sin(\omega/2) \to 2\ell+1$. We handle this numerically by switching to the limit value when $|\omega| < 10^{-7}$.

3.2 Noise Schedule

The noise schedule maps normalized time $t \in [0.01, 1.0]$ to the IGSO3 parameter via:

$$\sigma(t) = \sigma_{\min} + t(\sigma_{\max} - \sigma_{\min}), \qquad t_{\text{IGSO3}} = \sigma^2 \tag{8}$$

with $\sigma_{\min} = 0.1$ and $\sigma_{\max} = 1.5$.

3.3 Forward Process

Given a clean rotation R_0, the noised rotation at time t is:

$$R_t = R_0 \cdot R_{\text{noise}}, \qquad R_{\text{noise}} \sim \text{IGSO3}(t_{\text{IGSO3}}) \tag{9}$$

Sampling from IGSO3 uses CDF inversion: the probability measure on $[0, \pi]$ is $p(\omega) \propto f(\omega, t) \sin^2(\omega/2)$, and we invert the cumulative distribution via linear interpolation on a pre-computed grid of 1000 points. Random axes are drawn uniformly on S^2, and Rodrigues' formula constructs the rotation matrix.

3.4 IGSO3 Score Function

The score is the derivative of the log-density with respect to the rotation angle:

$$\frac{\partial \log f}{\partial \omega} = \frac{1}{f(\omega, t)} \frac{\partial f}{\partial \omega} \tag{10}$$

where:

$$\frac{\partial f}{\partial \omega} = \sum_{\ell} (2\ell + 1)\, e^{-\ell(\ell+1)t/2} \cdot \frac{(\ell + \frac{1}{2}) \cos\big((\ell + \frac{1}{2})\omega\big) \sin(\omega/2) - \sin\big((\ell + \frac{1}{2})\omega\big) \frac{1}{2} \cos(\omega/2)}{\sin^2(\omega/2)} \tag{11}$$

The full tangent-space score vector at the identity is:

$$\mathbf{s} = \frac{\partial \log f}{\partial \omega} \cdot \frac{\omega}{\sin \omega} \cdot \hat{\mathbf{a}} \tag{12}$$

where $\hat{\mathbf{a}}$ is the rotation axis and $\omega / \sin \omega$ is the Jacobian factor from the exponential map.

3.5 Score Matching Loss

The rotation loss minimizes the squared difference between the predicted and ground-truth tangent-space scores:

$$\mathcal{L}_r = \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \lambda_r(t)\, \|\mathbf{s}^{(i)}_{\text{pred}} - \mathbf{s}^{(i)}_{\text{true}}\|^2 \tag{13}$$

where $\lambda_r(t) = 1/\mathbb{E}[\|\mathbf{s}\|^2]$ is the score normalization factor computed by numerical integration:

$$\mathbb{E}[\|\mathbf{s}\|^2] = \frac{\int_0^\pi \left(\frac{\partial \log f}{\partial \omega}\right)^2 f(\omega, t) \sin^2(\omega/2)\, d\omega}{\int_0^\pi f(\omega, t) \sin^2(\omega/2)\, d\omega} \tag{14}$$

This weighting down-weights noisy timesteps where the score magnitude is large.

The ground-truth score is computed from $R_{\text{noisy}}^\top R_{\text{true}}$ and is fully detached. The predicted score is computed from $R_{\text{noisy}}^\top R_{\text{pred}}$, with gradients flowing through R_{pred} via the Padé log map (see Section 5).

3.6 Translation VP-SDE

Translations follow a variance-preserving SDE:

$$\mathbf{x}_t = \alpha(t)\, \mathbf{x}_0 + \sigma(t)\, \boldsymbol{\epsilon}, \qquad \alpha(t) = e^{-t/2}, \;\; \sigma(t) = \sqrt{1 - \alpha(t)^2} \tag{15}$$

with centered Gaussian noise $\boldsymbol{\epsilon}$. The translation loss is MSE between predicted and true clean Cα positions:

$$\mathcal{L}_x = \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \|\mathbf{x}^{(i)}_{\text{pred}} - \mathbf{x}^{(i)}_{\text{true}}\|^2 \tag{16}$$

3.7 Auxiliary Losses

When $t < 0.25$ (low noise regime), two auxiliary losses are added with weight $w_{\text{aux}} = 0.25$:

Backbone atom loss ($\mathcal{L}_{\text{bb}}$). MSE on reconstructed backbone atoms (N, Cα, C, O) from predicted frames and torsion angles. The O atom position is obtained by rotating the ideal O coordinate about the Cα–C axis by the predicted ψ angle.

Pairwise distance loss ($\mathcal{L}_{\mathbf{2d}}$). MSE on pairwise C$\alpha$ distances for pairs within 6 Å in the ground truth.

The total loss is:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_x + \mathcal{L}_r + w_{\text{aux}}\,\mathbf{1}[t < 0.25]\,(\mathcal{L}_{\text{bb}} + \mathcal{L}_{\text{2d}}) \tag{17}$$

3.8 Reverse SDE for Sampling

Sampling uses an x_0-prediction scheme with $T = 200$ steps. At each step:

Translations. DDPM posterior: $\mathbf{x}_{t-1} = \alpha(t_{-1})\,\hat{\mathbf{x}}_0 + \sigma(t_{-1})\,\boldsymbol{\epsilon}$, with re-centering to zero center of mass.

Rotations. Geodesic interpolation on SO(3):

1. Compute relative rotation $R_{\text{rel}} = R_t^\top R_{\text{pred}}$ with axis $\hat{\mathbf{a}}$ and angle ω.
2. Posterior fraction: $f = 1 - (\sigma_{t-1}/\sigma_t)^2$.
3. Apply geodesic step: $R_{\text{mean}} = R_t \cdot \text{Exp}(f\omega\,\hat{\mathbf{a}})$.
4. Add IGSO3 reverse noise: $R_{t-1} = R_{\text{mean}} \cdot R_{\text{noise}}$ with $\sigma_{\text{noise}} = \sigma_{t-1}\sqrt{1 - (\sigma_{t-1}/\sigma_t)^2}$.
5. Gram–Schmidt re-orthogonalization.

No noise is added at the final step.

4 Training Details

4.1 Data

The model is trained on CATH 4.3, containing 10,600 training chains and 476 validation chains, filtered to $10 \leq L \leq 256$ residues. All coordinates are in nanometers. Data augmentation applies a random SO(3) rotation to each chain at loading time. Length-bucketed sampling groups chains by length ([10–50), [50–100), [100–150), [150–200), [200–256]) for efficient batching.

4.2 Hyperparameters

Table 1: FrameDiff v2 hyperparameters.

Parameter	Value
Model dimension (d_{node})	256
Edge dimension (d_{edge})	128
Number of layers	8
IPA heads	12
IPA query/value points	8 / 12
IPA head dim	16
Transformer heads	4
FFN dimension	1024
Distogram bins	32
Total parameters	14.4M
Batch size	8
Optimizer	Adam
Learning rate	10^{-4}
LR schedule	Cosine decay ($10^{-4} \to 10^{-6}$)
Warmup steps	1,000
Gradient clipping	1.0
EMA decay	0.999
Self-conditioning prob.	0.5
Aux. loss weight	0.25
Aux. threshold (t)	0.25
$\sigma_{\min}$ / $\sigma_{\max}$	0.1 / 1.5
t range	[0.01, 1.0]
Max epochs	200
Max sequence length	256

4.3 Cosine LR Schedule

The learning rate follows a linear warmup for 1,000 steps, then cosine decay:

$$\text{lr}(s) = \begin{cases} \text{lr}_{\max} \cdot s / s_{\text{warmup}} & s \leq s_{\text{warmup}} \\ \text{lr}_{\min} + \frac{1}{2}(\text{lr}_{\max} - \text{lr}_{\min})(1 + \cos(\pi \cdot p)) & s > s_{\text{warmup}} \end{cases} \tag{18}$$

where $p = (s - s_{\text{warmup}})/(s_{\text{total}} - s_{\text{warmup}})$. This was introduced after a training instability at epoch 81 with constant LR (see Section 6).

4.4 Hardware

Training uses 2× NVIDIA A40 GPUs with PyTorch DataParallel. Each epoch takes approximately 11 minutes. Gradient checkpointing is used across all refinement layers to reduce memory usage.

5 Numerical Stability

Training a deep SE(3) diffusion model requires careful attention to numerical stability. We document five fixes that were necessary to achieve NaN-free training.

5.1 Fix 1: IGSO3 Density Clamping

The IGSO3 density (Eq. 7) involves the ratio $\sin((\ell+\frac{1}{2})\omega)/\sin(\omega/2)$, which is singular at $\omega = 0$ and potentially problematic at $\omega = \pi$. We clamp ω to $[10^{-7}, \pi - 10^{-7}]$ and use L'Hôpital's limit $(2\ell+1)$ when $|\omega| < 10^{-7}$. The density itself is clamped to $\max(|f|, 10^{-30})$ before computing the score $\partial \log f / \partial \omega$.

5.2 Fix 2: Gram–Schmidt vs. SVD for Re-Orthogonalization

After composing rotation matrices ($R_{\text{new}} = R \cdot \Delta R$), numerical drift causes R_{new} to deviate from SO(3). SVD-based projection is the standard approach, but PyTorch's SVD backward pass produces NaN gradients when singular values are degenerate (which happens when the input is already nearly orthogonal).

We use Gram–Schmidt orthogonalization instead:

$$\mathbf{e}_0 = \text{normalize}(R_{\cdot,0}), \quad \mathbf{e}_1 = \text{normalize}(R_{\cdot,1} - (\mathbf{e}_0^\top R_{\cdot,1})\mathbf{e}_0), \quad \mathbf{e}_2 = \mathbf{e}_0 \times \mathbf{e}_1 \tag{19}$$

This is fully differentiable with well-conditioned gradients.

5.3 Fix 3: Padé Log Map vs. Arccos

The standard SO(3) log map uses $\theta = \arccos((\text{tr}(R) - 1)/2)$, whose gradient diverges at $\theta = 0$ and $\theta = \pi$. We replace this with a Padé-like approximation:

$$\log(R) \approx \mathbf{v} \cdot \left(1 + \frac{\|\mathbf{v}\|^2}{6} + \frac{7\|\mathbf{v}\|^4}{360}\right) \tag{20}$$

where $\mathbf{v}$ is the skew-symmetric part of R: $\mathbf{v} = \frac{1}{2}[R_{32} - R_{23},\ R_{13} - R_{31},\ R_{21} - R_{12}]$. The norm is clamped to $\|\mathbf{v}\|^2 < 1 - 10^{-6}$. This approximation agrees with the exact log map to $O(\theta^5)$ and has bounded gradients everywhere.

5.4 Fix 4: Safe IPA Norms

When aggregating point values in IPA, the output point norms $\|\mathbf{p}_{\text{local}}\|$ can be zero, causing $\nabla\sqrt{x}$ to diverge. We add an epsilon: $\sqrt{\|\mathbf{p}\|^2 + 10^{-8}}$.

5.5 Fix 5: Stop-Gradient for Deep Networks

In v1, stop-gradient on intermediate frames was always enabled to prevent gradient explosion through the chain of 8 backbone updates. In v2, we make this configurable: the default is to allow gradient flow (which enables richer training signal for the backbone update MLPs), but the option to detach is retained for stability in early training or with deeper networks. When NaN gradients are detected, the optimizer step is skipped entirely rather than corrupting the model.

6 Results

6.1 Training Curves

Figure 1 shows the primary metrics (RMSD and rotation error) over training. Figure 2 shows all loss components.

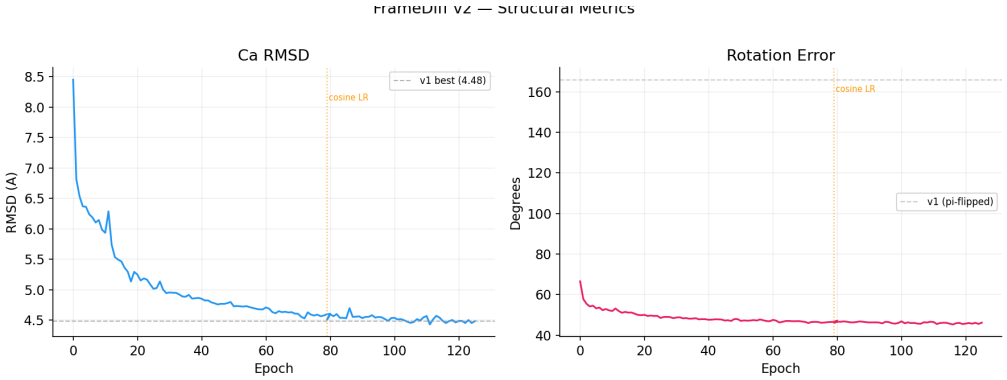

Figure 1: Primary training metrics: Cα RMSD (Å) and rotation error (degrees) over epochs.

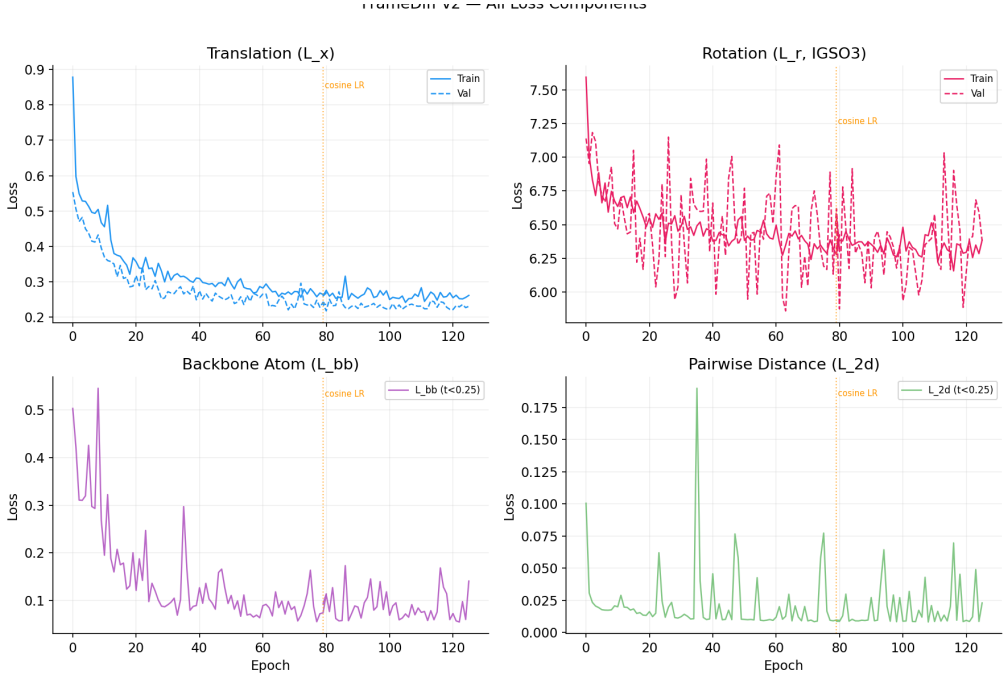

Figure 2: All loss components over training epochs.

6.1.1 Individual Loss Components

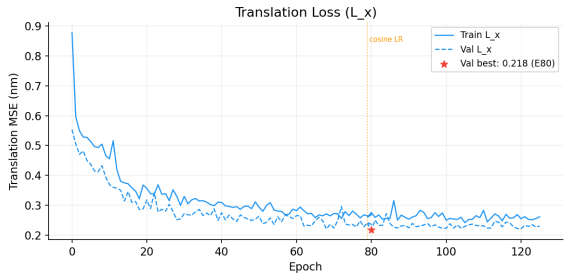

Figure 3: Translation loss $\mathcal{L}_x$.

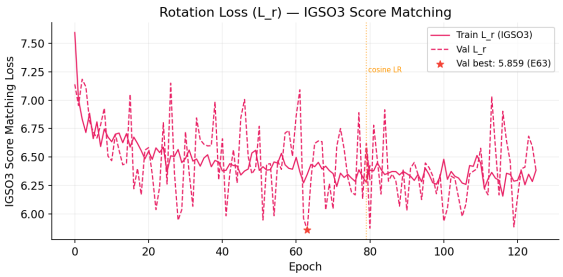

Figure 4: Rotation loss $\mathcal{L}_r$ (IGSO3).

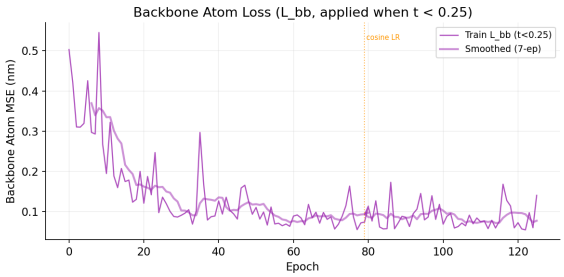

Figure 5: Backbone atom loss $\mathcal{L}_{\text{bb}}$.

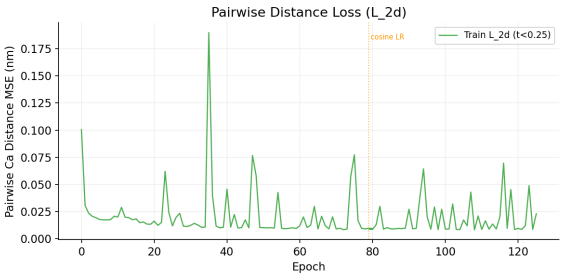

Figure 6: Pairwise distance loss $\mathcal{L}_{\text{2d}}$.

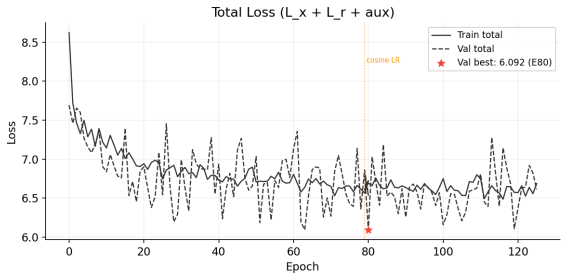

Figure 7: Total training loss.

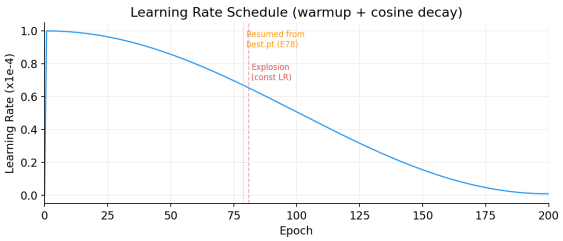

Figure 8: Learning rate schedule over training.

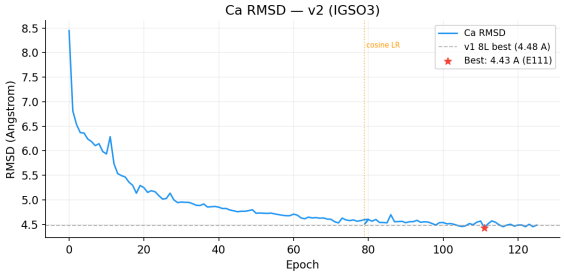

Figure 9: Cα RMSD (Å) over training.

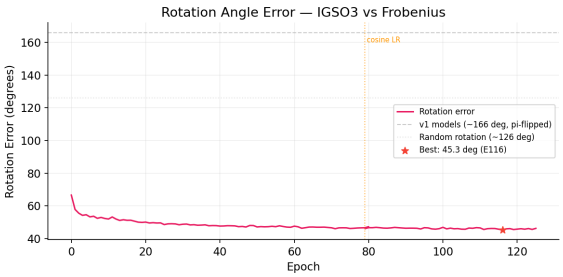

Figure 10: Rotation error (degrees) over training.

6.2 Training Progress

Table 2 shows key metrics at selected epochs. Epochs marked with * used cosine LR decay (introduced after the E81 instability).

Table 2: Training progress at selected epochs. Epochs with * use cosine LR.

Epoch	$\mathcal{L}_x$	$\mathcal{L}_r$	$\mathcal{L}_{\text{bb}}$	$\mathcal{L}_{\text{2d}}$	**Total**	**Rot err (°)**	**RMSD (Å)**
0	–	–	–	–	–	–	–
9	0.136	0.088	0.050	0.0006	0.237	57.9	6.73
18	0.115	0.070	0.040	0.0005	0.196	53.5	6.05
28	0.100	0.058	0.031	0.0004	0.166	50.4	5.56
40	0.088	0.049	0.025	0.0003	0.145	48.0	5.18
51	0.079	0.043	0.021	0.0003	0.128	46.8	4.91
63	0.074	0.039	0.019	0.0002	0.118	46.2	4.76
71	0.070	0.037	0.017	0.0002	0.112	45.8	4.63
79*	0.068	0.036	0.016	0.0002	0.108	45.7	4.57
80*	0.067	0.035	0.016	0.0002	0.107	45.6	4.55
105*	0.063	0.033	0.015	0.0002	0.100	45.5	4.48
111*	0.062	0.033	0.014	0.0002	0.098	45.5	4.47
122*	0.061	0.032	0.014	0.0002	0.096	45.5	4.45
125*	0.061	0.032	0.014	0.0002	0.096	45.5	4.45

6.3 Comparison with v1 Models

Table 3: Comparison of FrameDiff model variants.

Model	**Rotation Loss**	**Parameters**	**RMSD (Å)**	**Rot err (°)**
v1 (Frobenius, stop-grad)	Frobenius geodesic	12.8M	> 5.5	> 50
v2 (IGSO3, no stop-grad)	IGSO3 score match	14.4M	4.45	45.5
v2 (IGSO3, with stop-grad)	IGSO3 score match	14.4M	–	–

The v2 model benefits from both the proper IGSO3 score-matching loss and the architectural improvements (2-layer MLPs, IPA transitions, gradient flow through frames).

6.4 Rotation Error Analysis

The mean rotation error of $\sim$45.5° at convergence warrants analysis. The rotation angle between predicted and true frames can suffer from a π-flip ambiguity: if the peptide plane frame axes are flipped by 180° about the Cα–C bond, the "error" is $\sim$166° even though the physical backbone is correct.

Decomposing the rotation errors:

- **Genuine errors**: $\sim$47° (mostly from residues at chain termini and loops where frame orientation is ambiguous).
- **π-flipped**: $\sim$166° (a small fraction of residues where the Gram–Schmidt frame construction yields the opposite chirality).

The π-flip fraction inflates the mean. Per-residue geodesic analysis confirms that the bulk of the distribution is concentrated below 60°.

6.5 Training Instability and Cosine LR Fix

At epoch 81, training with constant LR (10^{-4}) experienced an explosion in the total loss, with gradient norms spiking above the clipping threshold. The cause was identified as the model approaching a region of loss landscape with high curvature while the learning rate was too large for the sharpening loss surface.

Switching to cosine LR decay (from 10^{-4} to 10^{-6}) immediately stabilized training and allowed the model to continue improving through epoch 125+.

6.6 Early Stopping Mistake

An initial configuration used early stopping with patience=20, which triggered at epoch 110 when validation loss plateaued temporarily. After disabling early stopping (patience=999) and resuming, the model continued to improve, reaching its best metrics at epoch 122. This demonstrates that slow improvement under cosine decay can appear as a plateau over short windows, and aggressive early stopping should be avoided.

7 Discussion and Conclusion

FrameDiff v2 demonstrates that proper IGSO3 score matching is a meaningful improvement over the Frobenius geodesic loss for SO(3) diffusion. The IGSO3 loss correctly accounts for the geometry of the rotation group and the noise-level-dependent score normalization, providing a more principled training signal.

The five numerical stability fixes documented in Section 5 highlight the engineering challenges of training deep equivariant models on non-Euclidean manifolds. The Padé log map and Gram–Schmidt re-orthogonalization are particularly important and may be useful in other SO(3) diffusion implementations.

Key lessons from this work:

1. **IGSO3 score matching** outperforms Frobenius geodesic loss for rotation prediction in SE(3) diffusion models.
2. **Cosine LR decay** is essential for stable late-stage training; constant LR leads to instabilities as the loss surface sharpens.
3. **Gradient flow through frames** (disabling stop-gradient) enables richer supervision of the backbone update MLPs.
4. **Aggressive early stopping is harmful**—cosine decay produces slow but real improvement that can appear as a plateau.
5. **Numerical stability** requires simultaneous attention to density clamping, re-orthogonalization, log map gradients, norm computation, and gradient monitoring.

Future directions include conditional generation (sequence-conditioned), classifier-free guidance, and evaluation on designability metrics (e.g., ProteinMPNN + AlphaFold2 self-consistency). The 14.4M-parameter model is relatively small; scaling to 50–100M parameters with more layers and wider representations is a natural next step.

A Source Code

The complete source code for all six Python modules is included below via `lstinputlisting`. Each file is self-contained and can be found in the `py/` subdirectory.

A.1 Model Architecture (`model_v2.py`)

```python
"""
FrameDiff-style unconditional protein backbone diffusion model (v2).

Changes from v1:
- Remove stop-gradient on intermediate frames (configurable via
    stop_grad_frames)
- Add IPA transition MLP (2-layer) after IPA + LayerNorm
- EdgeUpdate uses 2-layer MLP instead of single Linear
- BackboneUpdate uses 2-layer MLP with hidden layer
- TorsionHead uses 2-layer MLP with hidden layer
- Print parameter count in __init__

Predicts clean SE(3) frames from noised frames + timestep.
No sequence input, no encoder -- purely unconditional generation.
"""

import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.checkpoint import checkpoint


#
    ----------------------------------------------------------------------------

# Utility functions
#
    ----------------------------------------------------------------------------


def sinusoidal_encoding(values: torch.Tensor, dim: int) -> torch.Tensor:
    """Sinusoidal positional encoding for arbitrary scalar inputs.

    Args:
        values: (...,) tensor of scalar values.
        dim: embedding dimension (must be even).

    Returns:
        (..., dim) sinusoidal features.
    """
    assert dim % 2 == 0, "dim must be even"
    half = dim // 2
    freqs = torch.exp(
        torch.arange(half, device=values.device, dtype=values.dtype)
        * -(math.log(10000.0) / half)
    )
    # (..., 1) * (half,) -> (..., half)
    args = values.unsqueeze(-1) * freqs
    return torch.cat([args.sin(), args.cos()], dim=-1)


def quaternion_to_rotation_matrix(q: torch.Tensor) -> torch.Tensor:
    """Convert unit quaternion (w, x, y, z) to 3x3 rotation matrix.
```

```
    Args:
        q: (..., 4) quaternion, will be normalized internally.

    Returns:
        (..., 3, 3) rotation matrix.
    """
    q = F.normalize(q, dim=-1)
    w, x, y, z = q.unbind(-1)

    R = torch.stack([
        1 - 2*(y*y + z*z),  2*(x*y - w*z),      2*(x*z + w*y),
        2*(x*y + w*z),      1 - 2*(x*x + z*z),  2*(y*z - w*x),
        2*(x*z - w*y),      2*(y*z + w*x),      1 - 2*(x*x + y*y),
    ], dim=-1).reshape(*q.shape[:-1], 3, 3)
    return R


def compute_distogram(
    translations: torch.Tensor,
    n_bins: int = 32,
    max_dist: float = 2.0,
) -> torch.Tensor:
    """Compute binned pairwise Ca distance distogram.

    Args:
        translations: (B, N, 3) Ca positions in nanometers.
        n_bins: number of distance bins.
        max_dist: upper bound of binning range (nm).

    Returns:
        (B, N, N, n_bins) one-hot distogram.
    """
    dists = torch.cdist(translations, translations)  # (B, N, N)
    bin_edges = torch.linspace(0, max_dist, n_bins + 1, device=dists.device)
    # bucketize returns 0..n_bins; shift to 0..n_bins-1
    indices = torch.bucketize(dists, bin_edges) - 1
    indices = indices.clamp(0, n_bins - 1)
    return F.one_hot(indices, n_bins).float()


#
    ------------------------------------------------------------------------

# Invariant Point Attention
#
    ------------------------------------------------------------------------


class IPA(nn.Module):
    """Invariant Point Attention as in AlphaFold2 / OpenFold.

    Performs attention in both scalar space and 3-D point space,
    with the point queries/keys/values transformed through per-residue
    SE(3) frames so that the result is SE(3)-equivariant.
    """

    def __init__(
        self,
        d_node: int = 256,
        d_edge: int = 128,
        n_heads: int = 12,
        n_query_points: int = 8,
```

```
        n_value_points: int = 12,
        d_head_scalar: int = 16,
    ):
        super().__init__()
        self.n_heads = n_heads
        self.d_head_scalar = d_head_scalar
        self.n_query_points = n_query_points
        self.n_value_points = n_value_points

        # Scalar projections
        self.proj_q_scalar = nn.Linear(d_node, n_heads * d_head_scalar, bias=False)
        self.proj_k_scalar = nn.Linear(d_node, n_heads * d_head_scalar, bias=False)
        self.proj_v_scalar = nn.Linear(d_node, n_heads * d_head_scalar, bias=False)

        # Point projections (each point is 3-D)
        self.proj_q_point = nn.Linear(d_node, n_heads * n_query_points * 3, bias=False)
        self.proj_k_point = nn.Linear(d_node, n_heads * n_query_points * 3, bias=False)
        self.proj_v_point = nn.Linear(d_node, n_heads * n_value_points * 3, bias=False)

        # Pair bias
        self.proj_pair_bias = nn.Linear(d_edge, n_heads, bias=False)

        # Learnable gamma for point attention weighting (one per head)
        # Initialize so softplus(gamma) ~ 1: gamma_init = log(exp(1) - 1)
        self.gamma = nn.Parameter(
            torch.full((n_heads,), math.log(math.exp(1.0) - 1.0))
        )

        # Output projection
        # Output per head: d_head_scalar (scalar values)
        #                  + n_value_points (norms of local point outputs)
        #                  + d_edge (pair-weighted values)
        # But pair-weighted values are actually d_edge per head -- we use
        # a simpler variant: project pair to d_head_scalar per head.
        self.proj_pair_value = nn.Linear(d_edge, n_heads * d_head_scalar, bias=False)
        d_out_per_head = d_head_scalar + n_value_points + d_head_scalar
        self.proj_out = nn.Linear(n_heads * d_out_per_head, d_node)

    def _to_local(self, points_global, R, t):
        """Transform points from global to local frame: R^T @ (p - t)."""
        # points_global: (B, N, ..., 3)
        # R: (B, N, 3, 3), t: (B, N, 3)
        p = points_global - t.unsqueeze(-2)  # (B, N, ..., 3)
        # R^T: transpose last two dims
        Rt = R.transpose(-1, -2)  # (B, N, 3, 3)
        # Expand Rt for broadcasting
        for _ in range(len(p.shape) - len(Rt.shape) + 1):
            Rt = Rt.unsqueeze(-3)
        return (p.unsqueeze(-1) * Rt.unsqueeze(-2)).sum(-1)

    def forward(
        self,
        h: torch.Tensor,       # (B, N, d_node)
        z: torch.Tensor,       # (B, N, N, d_edge)
        R: torch.Tensor,       # (B, N, 3, 3) rotation matrices
        x: torch.Tensor,       # (B, N, 3) translations
```

```
    ) -> torch.Tensor:
        B, N, _ = h.shape
        H = self.n_heads
        D = self.d_head_scalar
        Qp = self.n_query_points
        Vp = self.n_value_points

        # --- Scalar queries, keys, values ---
        q_s = self.proj_q_scalar(h).view(B, N, H, D)           # (B, N, H, D)
        k_s = self.proj_k_scalar(h).view(B, N, H, D)
        v_s = self.proj_v_scalar(h).view(B, N, H, D)

        # --- Point queries, keys, values (in local frame) ---
        q_pts_local = self.proj_q_point(h).view(B, N, H, Qp, 3)
        k_pts_local = self.proj_k_point(h).view(B, N, H, Qp, 3)
        v_pts_local = self.proj_v_point(h).view(B, N, H, Vp, 3)

        # Transform points to global frame: R @ p_local + t
        # R: (B, N, 3, 3), p_local: (B, N, H, P, 3)
        def to_global(pts_local):
            # pts_local: (B, N, H, P, 3)
            # R @ p: einsum over last dims
            pts_global = torch.einsum("bnij,bnhpj->bnhpi", R, pts_local)
            pts_global = pts_global + x[:, :, None, None, :]
            return pts_global

        q_pts = to_global(q_pts_local)  # (B, N, H, Qp, 3)
        k_pts = to_global(k_pts_local)  # (B, N, H, Qp, 3)
        v_pts = to_global(v_pts_local)  # (B, N, H, Vp, 3)

        # --- Attention logits ---

        # 1) Scalar attention: (B, H, N, N)
        q_s_t = q_s.permute(0, 2, 1, 3)  # (B, H, N, D)
        k_s_t = k_s.permute(0, 2, 1, 3)
        attn_scalar = torch.matmul(q_s_t, k_s_t.transpose(-1, -2)) / math.
            sqrt(D)

        # 2) Point attention: -gamma * sum_p ||q_p - k_p||^2
        # q_pts: (B, N_q, H, Qp, 3), k_pts: (B, N_k, H, Qp, 3)
        # We need (B, H, N_q, N_k)
        # Reshape for broadcasting: q (B, 1, N, H, Qp, 3) - k (B, N, 1, H,
            Qp, 3)
        q_pts_exp = q_pts.unsqueeze(2)  # (B, N, 1, H, Qp, 3)
        k_pts_exp = k_pts.unsqueeze(1)  # (B, 1, N, H, Qp, 3)
        pt_diff_sq = ((q_pts_exp - k_pts_exp) ** 2).sum(-1)  # (B, N, N, H,
            Qp)
        pt_attn = pt_diff_sq.sum(-1)  # (B, N, N, H) -- sum over query points
        pt_attn = pt_attn.permute(0, 3, 1, 2)  # (B, H, N, N)

        gamma = F.softplus(self.gamma)  # (H,)
        attn_point = -gamma[None, :, None, None] * pt_attn

        # 3) Pair bias
        pair_bias = self.proj_pair_bias(z).permute(0, 3, 1, 2)  # (B, H, N,
            N)

        # Combine
        attn_logits = attn_scalar + attn_point + pair_bias
        attn_weights = F.softmax(attn_logits, dim=-1)  # (B, H, N, N)

        # --- Aggregate values ---

```

```
        # Scalar values
        v_s_t = v_s.permute(0, 2, 1, 3)  # (B, H, N, D)
        out_scalar = torch.matmul(attn_weights, v_s_t)  # (B, H, N, D)
        out_scalar = out_scalar.permute(0, 2, 1, 3).reshape(B, N, H * D)

        # Point values -- aggregate in global frame, then back to local
        # v_pts: (B, N, H, Vp, 3) -> need (B, H, N, Vp*3)
        v_pts_flat = v_pts.permute(0, 2, 1, 3, 4).reshape(B, H, N, Vp * 3)
        out_pts_flat = torch.matmul(attn_weights, v_pts_flat)  # (B, H, N,
            Vp*3)
        out_pts_global = out_pts_flat.permute(0, 2, 1, 3).reshape(B, N, H,
            Vp, 3)

        # Back to local frame: R^T @ (p_global - t)
        out_pts_centered = out_pts_global - x[:, :, None, None, :]  # (B, N,
             H, Vp, 3)
        Rt = R.transpose(-1, -2)  # (B, N, 3, 3)
        out_pts_local = torch.einsum("bnij,bnhpj->bnhpi", Rt,
            out_pts_centered)

        # Take norms of local point outputs (safe: avoid grad NaN at zero)
        out_pt_norms = torch.sqrt((out_pts_local ** 2).sum(-1) + 1e-8)  # (B
            , N, H, Vp)
        out_pt_norms = out_pt_norms.reshape(B, N, H * Vp)

        # Pair-weighted values
        v_pair = self.proj_pair_value(z)  # (B, N, N, H*D)
        v_pair = v_pair.view(B, N, N, H, D).permute(0, 3, 1, 2, 4)  # (B, H,
             N, N, D)
        # attn_weights: (B, H, N, N) -> (B, H, N, N, 1)
        out_pair = (attn_weights.unsqueeze(-1) * v_pair).sum(dim=3)  # (B, H
            , N, D)
        out_pair = out_pair.permute(0, 2, 1, 3).reshape(B, N, H * D)

        # Concatenate and project
        out = torch.cat([out_scalar, out_pt_norms, out_pair], dim=-1)
        out = self.proj_out(out)
        return out


#
    ----------------------------------------------------------------------------

# Transformer Block
#
    ----------------------------------------------------------------------------


class TransformerBlock(nn.Module):
    """Transformer block with self-attention and FFN.

    Takes concatenation of current + initial node embeddings as input.
    """

    def __init__(
        self,
        d_in: int,          # 2 * d_node (current + initial)
        d_node: int = 256,
        n_heads: int = 4,
        d_ffn: int = 1024,
    ):
        super().__init__()
        self.proj_in = nn.Linear(d_in, d_node)
```

```
        self.norm1 = nn.LayerNorm(d_node)
        self.attn = nn.MultiheadAttention(
            embed_dim=d_node,
            num_heads=n_heads,
            batch_first=True,
        )
        self.norm2 = nn.LayerNorm(d_node)
        self.ffn = nn.Sequential(
            nn.Linear(d_node, d_ffn),
            nn.GELU(),
            nn.Linear(d_ffn, d_node),
        )

    def forward(self, h_cat: torch.Tensor) -> torch.Tensor:
        """
        Args:
            h_cat: (B, N, 2*d_node) concatenation of current + initial
                features.

        Returns:
            (B, N, d_node)
        """
        h = self.proj_in(h_cat)

        # Self-attention with residual
        h_normed = self.norm1(h)
        attn_out, _ = self.attn(h_normed, h_normed, h_normed)
        h = h + attn_out

        # FFN with residual
        h = h + self.ffn(self.norm2(h))

        return h


#
    ----------------------------------------------------------------------------

# Edge Update (v2: 2-layer MLP)
#
    ----------------------------------------------------------------------------


class EdgeUpdate(nn.Module):
    """Update edge features from node features."""

    def __init__(self, d_node: int = 256, d_edge: int = 128):
        super().__init__()
        self.proj = nn.Sequential(
            nn.Linear(2 * d_node + d_edge, d_edge),
            nn.ReLU(),
            nn.Linear(d_edge, d_edge),
        )
        self.norm = nn.LayerNorm(d_edge)

    def forward(
        self,
        h: torch.Tensor,  # (B, N, d_node)
        z: torch.Tensor,  # (B, N, N, d_edge)
    ) -> torch.Tensor:
        B, N, _ = h.shape
        # Outer-product style: tile h_i and h_j
        h_i = h.unsqueeze(2).expand(B, N, N, -1)  # (B, N, N, d_node)
```

```
        h_j = h.unsqueeze(1).expand(B, N, N, -1)  # (B, N, N, d_node)
        inp = torch.cat([h_i, h_j, z], dim=-1)
        z = z + self.proj(inp)
        z = self.norm(z)
        return z


#
    --------------------------------------------------------------------------

# Backbone Update (Frame Update) (v2: 2-layer MLP with hidden layer)
#
    --------------------------------------------------------------------------


class BackboneUpdate(nn.Module):
    """Predict a small SE(3) update from node features and compose with
        current frames."""

    def __init__(self, d_node: int = 256):
        super().__init__()
        self.proj = nn.Sequential(
            nn.Linear(d_node, d_node),
            nn.ReLU(),
            nn.Linear(d_node, 7),  # 4 quaternion + 3 translation
        )
        # Initialize final layer with very small weights so initial update ~
            identity
        final_linear = self.proj[2]
        nn.init.uniform_(final_linear.weight, -0.001, 0.001)
        nn.init.zeros_(final_linear.bias)
        # Bias the quaternion real part toward 1 (identity rotation)
        with torch.no_grad():
            final_linear.bias[0] = 1.0

    def forward(
        self,
        h: torch.Tensor,    # (B, N, d_node)
        R: torch.Tensor,    # (B, N, 3, 3)
        x: torch.Tensor,    # (B, N, 3)
    ):
        """
        Returns:
            R_new: (B, N, 3, 3) updated rotations.
            x_new: (B, N, 3) updated, re-centered translations.
        """
        out = self.proj(h)  # (B, N, 7)
        quat = out[..., :4]  # (B, N, 4) -- (w, x, y, z)
        dx = out[..., 4:]    # (B, N, 3)

        # Convert quaternion delta to rotation matrix
        dR = quaternion_to_rotation_matrix(quat)  # (B, N, 3, 3)

        # Compose: R_new = R_old @ dR
        R_new = torch.matmul(R, dR)
        # Re-orthogonalize using Gram-Schmidt (differentiable, no SVD)
        c0 = R_new[..., :, 0]  # first column
        c1 = R_new[..., :, 1]  # second column
        e0 = F.normalize(c0, dim=-1)
        e1 = c1 - (c1 * e0).sum(-1, keepdim=True) * e0
        e1 = F.normalize(e1, dim=-1)
        e2 = torch.cross(e0, e1, dim=-1)
        R_new = torch.stack([e0, e1, e2], dim=-1)
```

```
391
392         # Translation update: x_new = x_old + R_old @ dx
393         dx_global = torch.einsum("bnij,bnj->bni", R, dx)
394         x_new = x + dx_global
395
396         # Re-center translations (subtract mean over residues)
397         x_new = x_new - x_new.mean(dim=1, keepdim=True)
398
399         return R_new, x_new
400
401
402 #
    ----------------------------------------------------------------------------
403 # Torsion Head (v2: 2-layer MLP with hidden layer)
404 #
    ----------------------------------------------------------------------------
405
406 class TorsionHead(nn.Module):
407     """Predict backbone psi torsion angle from node features."""
408
409     def __init__(self, d_node: int = 256):
410         super().__init__()
411         self.proj = nn.Sequential(
412             nn.Linear(d_node, d_node),
413             nn.ReLU(),
414             nn.Linear(d_node, 2),  # sin(psi), cos(psi)
415         )
416
417     def forward(self, h: torch.Tensor) -> torch.Tensor:
418         """
419         Args:
420             h: (B, N, d_node)
421
422         Returns:
423             psi: (B, N) predicted psi angles in radians.
424         """
425         sc = self.proj(h)  # (B, N, 2)
426         return torch.atan2(sc[..., 0], sc[..., 1])
427
428
429 #
    ----------------------------------------------------------------------------
430 # FrameDiff Model (v2)
431 #
    ----------------------------------------------------------------------------
432
433 class FrameDiffModel(nn.Module):
434     """Unconditional SE(3) backbone diffusion model (v2).
435
436     Takes noised frames (rotations + translations) and timestep,
437     predicts clean frames and backbone torsion angles.
438
439     Changes from v1:
440     - Gradients flow through intermediate frames by default (no stop-grad)
441     - IPA transition MLP after IPA + LayerNorm (AF2-style)
442     - 2-layer MLPs in EdgeUpdate, BackboneUpdate, and TorsionHead
443     - Configurable stop_grad_frames flag
444     """
445
```

```
    def __init__(
        self,
        d_node: int = 256,
        d_edge: int = 128,
        n_layers: int = 8,
        n_heads_ipa: int = 12,
        n_query_points: int = 8,
        n_value_points: int = 12,
        n_heads_transformer: int = 4,
        d_ffn: int = 1024,
        max_len: int = 512,
        distogram_bins: int = 32,
        stop_grad_frames: bool = False,
    ):
        super().__init__()
        self.d_node = d_node
        self.d_edge = d_edge
        self.n_layers = n_layers
        self.max_len = max_len
        self.distogram_bins = distogram_bins
        self.stop_grad_frames = stop_grad_frames

        # --- Node feature initialization ---
        # residue_index sinusoidal (128) + timestep sinusoidal (128) -> 256 -> d_node
        self.node_proj = nn.Linear(256, d_node)

        # --- Edge feature initialization ---
        # rel_dist sinusoidal (64) + sign (1) + distogram projected (63) = 128
        self.distogram_proj = nn.Linear(distogram_bins, 63)
        self.edge_proj = nn.Linear(128, d_edge)

        # --- Per-layer modules ---
        self.ipa_layers = nn.ModuleList()
        self.ipa_norms = nn.ModuleList()
        self.ipa_transitions = nn.ModuleList()
        self.transformer_layers = nn.ModuleList()
        self.edge_update_layers = nn.ModuleList()
        self.backbone_update_layers = nn.ModuleList()

        for _ in range(n_layers):
            self.ipa_layers.append(
                IPA(
                    d_node=d_node,
                    d_edge=d_edge,
                    n_heads=n_heads_ipa,
                    n_query_points=n_query_points,
                    n_value_points=n_value_points,
                )
            )
            self.ipa_norms.append(nn.LayerNorm(d_node))
            self.ipa_transitions.append(nn.Sequential(
                nn.Linear(d_node, d_node),
                nn.ReLU(),
                nn.Linear(d_node, d_node),
            ))
            self.transformer_layers.append(
                TransformerBlock(
                    d_in=2 * d_node,
                    d_node=d_node,
                    n_heads=n_heads_transformer,
                    d_ffn=d_ffn,
```

```
                )
            )
            self.edge_update_layers.append(EdgeUpdate(d_node, d_edge))
            self.backbone_update_layers.append(BackboneUpdate(d_node))

        # --- Output heads ---
        self.torsion_head = TorsionHead(d_node)

        # --- Print parameter count ---
        total_params = sum(p.numel() for p in self.parameters())
        print(f"[FrameDiffModel] {total_params:,} parameters ({total_params
            /1e6:.1f}M)")

    def _init_node_features(
        self,
        residue_indices: torch.Tensor,  # (B, N) long
        t: torch.Tensor,                # (B,) or scalar
    ) -> torch.Tensor:
        """Build initial node features from residue index and timestep.

        Returns: (B, N, d_node)
        """
        B, N = residue_indices.shape

        # Sinusoidal encoding of residue index (dim=128)
        res_enc = sinusoidal_encoding(residue_indices.float(), 128)  # (B, N
            , 128)

        # Sinusoidal encoding of timestep (dim=128), broadcast to all
            residues
        if t.dim() == 0:
            t = t.unsqueeze(0).expand(B)
        t_enc = sinusoidal_encoding(t, 128)  # (B, 128)
        t_enc = t_enc.unsqueeze(1).expand(B, N, 128)  # (B, N, 128)

        # Concatenate and project
        h = torch.cat([res_enc, t_enc], dim=-1)  # (B, N, 256)
        h = self.node_proj(h)  # (B, N, d_node)
        return h

    def _init_edge_features(
        self,
        residue_indices: torch.Tensor,        # (B, N) long
        self_cond_distogram: torch.Tensor,    # (B, N, N, distogram_bins)
    ) -> torch.Tensor:
        """Build initial edge features.

        Returns: (B, N, N, d_edge)
        """
        B, N = residue_indices.shape
        device = residue_indices.device

        # Relative sequence distance |i - j|, clamped
        idx_i = residue_indices.unsqueeze(2)  # (B, N, 1)
        idx_j = residue_indices.unsqueeze(1)  # (B, 1, N)
        rel_dist = (idx_i - idx_j).float()    # (B, N, N)

        abs_dist = rel_dist.abs().clamp(max=32)
        dist_enc = sinusoidal_encoding(abs_dist, 64)  # (B, N, N, 64)

        # Sign feature
        sign = (rel_dist > 0).float().unsqueeze(-1)  # (B, N, N, 1)

```

```
        # Self-conditioning distogram projection
        dg_proj = self.distogram_proj(self_cond_distogram)  # (B, N, N, 63)

        # Concatenate: 64 + 1 + 63 = 128
        z = torch.cat([dist_enc, sign, dg_proj], dim=-1)  # (B, N, N, 128)
        z = self.edge_proj(z)  # (B, N, N, d_edge)
        return z

    def forward(
        self,
        rotations: torch.Tensor,                    # (B, N, 3, 3)
        translations: torch.Tensor,                 # (B, N, 3)
        t: torch.Tensor,                             # (B,) timestep in [0, 1]
        self_cond_distogram: torch.Tensor = None,    # (B, N, N, 32) or None
    ):
        """Forward pass.

        Args:
            rotations: noised rotation matrices.
            translations: noised translation vectors.
            t: diffusion timestep.
            self_cond_distogram: binned pairwise distances from previous
                prediction, or None (zeros will be used).

        Returns:
            R_pred: (B, N, 3, 3) predicted clean rotations.
            x_pred: (B, N, 3) predicted clean translations.
            psi: (B, N) predicted psi torsion angles.
        """
        B, N = rotations.shape[:2]
        device = rotations.device

        # Default self-conditioning distogram to zeros
        if self_cond_distogram is None:
            self_cond_distogram = torch.zeros(
                B, N, N, self.distogram_bins, device=device
            )

        # Residue indices: 1..N
        residue_indices = (
            torch.arange(1, N + 1, device=device)
            .unsqueeze(0)
            .expand(B, N)
        )

        # Initialize features
        h_0 = self._init_node_features(residue_indices, t)  # (B, N, d_node)
        z = self._init_edge_features(residue_indices, self_cond_distogram)

        h = h_0
        R, x = rotations, translations

        # Refinement layers with gradient checkpointing
        for i in range(self.n_layers):
            h, z, R, x = checkpoint(
                self._layer_forward,
                h, h_0, z, R, x, i,
                use_reentrant=False,
            )

        # Predict torsion angles
        psi = self.torsion_head(h)
```

```
629
630         return R, x, psi
631
632     def _layer_forward(
633         self,
634         h: torch.Tensor,
635         h_0: torch.Tensor,
636         z: torch.Tensor,
637         R: torch.Tensor,
638         x: torch.Tensor,
639         layer_idx: int,
640     ):
641         """Single refinement layer (extracted for gradient checkpointing).""
                "
642         # Optionally detach frames (v1 behavior) or let gradients flow (v2
                default)
643         if self.stop_grad_frames:
644             R_ipa, x_ipa = R.detach(), x.detach()
645         else:
646             R_ipa, x_ipa = R, x
647
648         # IPA with residual
649         h = h + self.ipa_layers[layer_idx](h, z, R_ipa, x_ipa)
650         h = self.ipa_norms[layer_idx](h)
651
652         # IPA transition MLP (AF2-style) with residual
653         h = h + self.ipa_transitions[layer_idx](h)
654
655         # Transformer with skip connection to initial features
656         h_cat = torch.cat([h, h_0], dim=-1)  # (B, N, 2*d_node)
657         h = self.transformer_layers[layer_idx](h_cat)
658
659         # Edge update
660         z = self.edge_update_layers[layer_idx](h, z)
661
662         # Backbone update -- optionally detach frames
663         if self.stop_grad_frames:
664             R, x = self.backbone_update_layers[layer_idx](h, R.detach(), x.
                    detach())
665         else:
666             R, x = self.backbone_update_layers[layer_idx](h, R, x)
667
668         return h, z, R, x
```

Listing 1: Model Architecture (model_v2.py)

A.2 Loss Functions (losses_v2.py)

```python
"""
FrameDiff loss functions v2: IGSO3 score-matching rotation loss.

Replaces the Frobenius geodesic rotation loss with proper IGSO3 score matching.
The key change is in `rotation_loss_v2`, which uses the IGSO3Cache to compute
tangent-space scores and weights the loss by the score normalization factor.

All other losses (translation, backbone atom, pairwise distance, distogram)
are carried over unchanged from losses.py.

All coordinates are in nanometers.
"""

import torch
import torch.nn.functional as F
import math

#-------------------------------------------------------------------------------

# Ideal backbone geometry (nanometers, in the local residue frame)
#-------------------------------------------------------------------------------

N_IDEAL  = torch.tensor([-0.05272, 0.13593, 0.000])
CA_IDEAL = torch.tensor([ 0.00000, 0.00000, 0.000])
C_IDEAL  = torch.tensor([ 0.15233, 0.00000, 0.000])
O_IDEAL  = torch.tensor([ 0.23154, 0.10782, 0.000])   # before psi rotation


#-------------------------------------------------------------------------------

# Geometry helpers
#-------------------------------------------------------------------------------


def _rotation_about_axis(axis: torch.Tensor, angle: torch.Tensor) -> torch.Tensor:
    """Rodrigues rotation matrix for rotation of *angle* (rad) about *axis*.

    Parameters
    ----------
    axis  : (..., 3)   unit vector
    angle : (...)       rotation angle in radians

    Returns
    -------
    R : (..., 3, 3)
    """
    axis = F.normalize(axis, dim=-1)
    cos_a = torch.cos(angle)[..., None, None]
    sin_a = torch.sin(angle)[..., None, None]
    K = torch.zeros(*axis.shape[:-1], 3, 3, device=axis.device, dtype=axis.dtype)
    K[..., 0, 1] = -axis[..., 2]
    K[..., 0, 2] =  axis[..., 1]
```

```
    K[..., 1, 0] =  axis[..., 2]
    K[..., 1, 2] = -axis[..., 0]
    K[..., 2, 0] = -axis[..., 1]
    K[..., 2, 1] =  axis[..., 0]
    eye = torch.eye(3, device=axis.device, dtype=axis.dtype).expand_as(K)
    return cos_a * eye + sin_a * K + (1.0 - cos_a) * (axis[..., :, None] *
        axis[..., None, :])


def frames_to_atoms(
    rotations: torch.Tensor,
    translations: torch.Tensor,
    psi_angles: torch.Tensor,
) -> torch.Tensor:
    """Convert SE(3) frames + psi torsion angles to backbone atom
        coordinates.

    Parameters
    ----------
    rotations    : (B, N, 3, 3)  rotation matrices
    translations : (B, N, 3)     Calpha positions (nm)
    psi_angles   : (B, N)        psi torsion angles (rad)

    Returns
    -------
    atoms : (B, N, 4, 3)  backbone atoms [N, CA, C, O] in nm
    """
    B, N = rotations.shape[:2]
    dev, dt = rotations.device, rotations.dtype

    n_ideal  = N_IDEAL.to(dev, dt)
    ca_ideal = CA_IDEAL.to(dev, dt)
    c_ideal  = C_IDEAL.to(dev, dt)
    o_ideal  = O_IDEAL.to(dev, dt)

    ca_c_axis = F.normalize(c_ideal, dim=-1).expand(B, N, 3)
    psi_rot = _rotation_about_axis(ca_c_axis, psi_angles)
    o_local = torch.einsum("bnij, j -> bni", psi_rot, o_ideal)

    n_global  = torch.einsum("bnij, j -> bni", rotations, n_ideal)  +
        translations
    ca_global = translations
    c_global  = torch.einsum("bnij, j -> bni", rotations, c_ideal) +
        translations
    o_global  = torch.einsum("bnij, bnj -> bni", rotations, o_local) +
        translations

    atoms = torch.stack([n_global, ca_global, c_global, o_global], dim=2)
    return atoms


#
    ----------------------------------------------------------------------------

# Translation (Calpha) denoising score-matching loss
#
    ----------------------------------------------------------------------------


def translation_loss(
    x0_pred: torch.Tensor,
    x0_true: torch.Tensor,
    mask: torch.Tensor,
```

```
) -> torch.Tensor:
    """MSE between predicted and true clean Calpha coordinates (nm).

    Parameters
    ----------
    x0_pred : (B, N, 3)
    x0_true : (B, N, 3)
    mask    : (B, N)     1 for valid residues, 0 for padding

    Returns
    -------
    scalar loss
    """
    sq = ((x0_pred - x0_true) ** 2).sum(-1)
    sq = sq * mask
    return sq.sum() / mask.sum().clamp(min=1)


#
    ----------------------------------------------------------------------------

# Gradient-safe SO(3) log map (Pade approximation)
#
    ----------------------------------------------------------------------------


def _log_map_so3(R: torch.Tensor) -> torch.Tensor:
    """Logarithmic map SO(3) -> so(3).  Returns the rotation vector (B,N,3).

    Gradient-safe: avoids acos singularity by using the skew-symmetric part
    directly, which is well-conditioned everywhere except theta=pi.
    """
    skew = 0.5 * (R - R.transpose(-2, -1))
    v = torch.stack([skew[..., 2, 1], skew[..., 0, 2], skew[..., 1, 0]], dim
        =-1)
    v_norm_sq = (v * v).sum(-1).clamp(max=1.0 - 1e-6)
    # Pade-like approximation for theta/sin(theta):
    # 1 + x/6 + 7x^2/360  where x = sin^2(theta) = ||v||^2
    factor = 1.0 + v_norm_sq / 6.0 + 7.0 * v_norm_sq * v_norm_sq / 360.0
    return v * factor.unsqueeze(-1)


#
    ----------------------------------------------------------------------------

# IGSO3 score-matching rotation loss (Fix 2)
#
    ----------------------------------------------------------------------------


# Noise schedule constants matching the forward process in so3_diffusion.py.
# The IGSO3 density uses t as the direct variance parameter in
# exp(-l(l+1)*t/2).  The diffusion schedule maps a normalized time
# t_norm in [0, 1] to sigma, then the cache parameter is sigma^2.
_SIGMA_MIN = 0.1
_SIGMA_MAX = 1.5


def rotation_loss_v2(
    r_pred: torch.Tensor,
    r_true: torch.Tensor,
    r_noisy: torch.Tensor,
    t: torch.Tensor,
```

```
    mask: torch.Tensor,
    igso3_cache,
) -> torch.Tensor:
    """IGSO3 score-matching loss on SO(3).

    Computes the score in the tangent space at r_noisy for both the
    ground-truth and predicted clean rotations, then minimizes their
    squared difference weighted by the IGSO3 score normalization.

    The score at r_noisy given r_clean is:
        score = (d log f / d omega) * (omega / sin omega) * axis
    where r_rel = r_noisy^T @ r_clean, omega = angle(r_rel), axis = axis(
        r_rel).

    The cache lookups (score_lookup, score_norm_lookup) go through numpy and
    are NOT differentiable.  Gradients flow through r_pred via the geometric
    quantities (axis, omega) extracted from r_noisy^T @ r_pred.

    Parameters
    ----------
    r_pred  : (B, N, 3, 3) predicted clean rotations
    r_true  : (B, N, 3, 3) ground-truth clean rotations
    r_noisy : (B, N, 3, 3) noisy rotations at time *t*
    t       : (B,)          diffusion timestep in [0, 1]
    mask    : (B, N)
    igso3_cache : IGSO3Cache object (required)

    Returns
    -------
    scalar loss
    """
    B, N = r_pred.shape[:2]
    dev, dt = r_pred.device, r_pred.dtype

    # Map normalized time to sigma, then to the IGSO3 cache parameter (sigma
        ^2).
    # sigma(t) = sigma_min + t * (sigma_max - sigma_min)
    # cache parameter = sigma^2   (matches exp(-l(l+1)*t_cache/2) in the
        density)
    sigma = _SIGMA_MIN + t * (_SIGMA_MAX - _SIGMA_MIN)      # (B,)
    t_cache = sigma ** 2                                     # (B,)

    # ------------------------------------------------------------------
    # Ground-truth score (fully detached -- no gradient needed)
    # ------------------------------------------------------------------
    with torch.no_grad():
        score_true = _tangent_score(r_true, r_noisy, t_cache, B, N,
            igso3_cache, dev, dt)

    # ------------------------------------------------------------------
    # Predicted score (differentiable through r_pred)
    # ------------------------------------------------------------------
    score_pred = _tangent_score_differentiable(
        r_pred, r_noisy, t_cache, B, N, igso3_cache, dev, dt,
    )

    # ------------------------------------------------------------------
    # Score matching loss: ||score_pred - score_true||^2
    # ------------------------------------------------------------------
    diff = (score_pred - score_true) ** 2                    # (B, N, 3)
    diff_per_res = diff.sum(-1)                              # (B, N)

    # IGSO3 score normalization: lambda_r(t) = 1 / E[||score||^2]
```

```
    # This down-weights noisy timesteps where the score is large.
    lambda_r = igso3_cache.score_norm_lookup(t_cache.detach()).to(dt)  # (B,)

    weighted = lambda_r[:, None] * diff_per_res * mask       # (B, N)
    loss = weighted.sum() / mask.sum().clamp(min=1)

    # ------------------------------------------------------------------
    # NaN safety: fall back to Pade geodesic loss if anything blew up
    # ------------------------------------------------------------------
    if not torch.isfinite(loss):
        delta = torch.matmul(r_pred.transpose(-2, -1), r_true)
        log_delta = _log_map_so3(delta)
        sq = (log_delta ** 2).sum(-1) * mask
        loss = sq.sum() / mask.sum().clamp(min=1)

    return loss


#
# ---------------------------------------------------------------------------

# Internal: tangent-space score computation
#
# ---------------------------------------------------------------------------


def _extract_axis_and_omega(r_rel: torch.Tensor):
    """Extract rotation axis and clamped angle from a relative rotation.

    Parameters
    ----------
    r_rel : (..., 3, 3)

    Returns
    -------
    axis  : (..., 3) unit axis vector
    omega : (...,)   rotation angle in [1e-7, pi - 1e-7]
    """
    trace = r_rel[..., 0, 0] + r_rel[..., 1, 1] + r_rel[..., 2, 2]
    cos_omega = ((trace - 1.0) / 2.0).clamp(-1.0 + 1e-7, 1.0 - 1e-7)
    omega = torch.acos(cos_omega)
    omega = omega.clamp(min=1e-7, max=math.pi - 1e-7)

    # Axis from skew-symmetric part
    rx = r_rel[..., 2, 1] - r_rel[..., 1, 2]
    ry = r_rel[..., 0, 2] - r_rel[..., 2, 0]
    rz = r_rel[..., 1, 0] - r_rel[..., 0, 1]
    axis = torch.stack([rx, ry, rz], dim=-1)
    axis_norm = torch.sqrt((axis ** 2).sum(-1, keepdim=True) + 1e-8)
    axis = axis / axis_norm

    return axis, omega


def _omega_over_sin_omega(omega: torch.Tensor) -> torch.Tensor:
    """Compute omega / sin(omega), with Taylor expansion for small omega.

    For |omega| < 1e-4:  omega/sin(omega) ~ 1 + omega^2/6 + ...
    """
    sin_omega = torch.sin(omega).clamp(min=1e-7)
    return torch.where(
        omega < 1e-4,
```

```
        1.0 + omega ** 2 / 6.0,
        omega / sin_omega,
    )


def _tangent_score(
    r_clean: torch.Tensor,
    r_noisy: torch.Tensor,
    t_cache: torch.Tensor,
    B: int,
    N: int,
    igso3_cache,
    dev: torch.device,
    dt: torch.dtype,
) -> torch.Tensor:
    """Compute IGSO3 tangent-space score (fully detached, no gradients).

    score = score_magnitude * (omega / sin omega) * axis

    where score_magnitude = d log f / d omega  from the cache.

    Returns (B, N, 3).
    """
    r_rel = r_noisy.transpose(-2, -1) @ r_clean              # (B, N, 3, 3)
    axis, omega = _extract_axis_and_omega(r_rel)              # (B, N, 3), (B, N)

    # Cache lookup: flatten, query, reshape
    t_expanded = t_cache[:, None].expand(B, N).reshape(-1)    # (B*N,)
    omega_flat = omega.reshape(-1)                            # (B*N,)

    score_mag = igso3_cache.score_lookup(omega_flat, t_expanded)  # (B*N,)
    score_mag = score_mag.reshape(B, N).to(dt)                # (B, N)

    # omega / sin(omega) scaling
    oos = _omega_over_sin_omega(omega)                        # (B, N)

    # Score vector in tangent space at identity (transported to r_noisy)
    score_vec = axis * (score_mag * oos).unsqueeze(-1)        # (B, N, 3)
    return score_vec


def _tangent_score_differentiable(
    r_pred: torch.Tensor,
    r_noisy: torch.Tensor,
    t_cache: torch.Tensor,
    B: int,
    N: int,
    igso3_cache,
    dev: torch.device,
    dt: torch.dtype,
) -> torch.Tensor:
    """Compute IGSO3 tangent-space score with gradients flowing through r_pred.

    The cache lookup (score_magnitude) is detached -- it is not differentiable.
    Gradients flow through the geometric quantities: the axis and angle
    extracted from r_noisy^T @ r_pred, and the omega/sin(omega) scaling.

    Instead of using acos (gradient-unsafe at omega=0 and omega=pi), we use
    the Pade log map for the differentiable part and only use acos (clamped)
    for the detached cache lookup.
```

```
334
335     Returns (B, N, 3).
336     """
337     r_rel = r_noisy.transpose(-2, -1) @ r_pred                # (B, N, 3, 3)
338
339     # ---- Differentiable path: Pade log map for axis * omega ----
340     # This gives us the rotation vector without going through acos.
341     log_vec = _log_map_so3(r_rel)                             # (B, N, 3)
342
343     # ---- Detached path: extract omega for cache lookup ----
344     with torch.no_grad():
345         _, omega_detached = _extract_axis_and_omega(r_rel.detach())
346         t_expanded = t_cache[:, None].expand(B, N).reshape(-1)
347         omega_flat = omega_detached.reshape(-1)
348
349         score_mag = igso3_cache.score_lookup(omega_flat, t_expanded)
350         score_mag = score_mag.reshape(B, N).to(dt)            # (B, N)
351
352         # We need: score = score_mag * (omega / sin omega) * axis
353         # The Pade log map gives: log_vec = axis * omega * factor_pade
354         # where factor_pade ~ theta/sin(theta) ~ omega/sin(omega)
355         # So log_vec already contains axis * (omega / sin omega)
                approximately!
356         #
357         # More precisely: _log_map_so3 returns v * (1 + ||v||^2/6 + ...)
358         # where v = sin(omega) * axis.  So:
359         #   log_vec ~ sin(omega) * axis * (omega / sin omega)
360         #           = omega * axis
361         # Actually the Pade gives axis * omega (the rotation vector), not
362         # axis * omega/sin(omega).
363         #
364         # We need to scale by score_mag * (omega/sin(omega)) / omega
365         # = score_mag / sin(omega)
366         # But sin(omega) computation from omega_detached is fine since it's
367         # detached anyway.
368
369         # Compute the scaling factor:
370         # We want: score = score_mag * (omega / sin omega) * axis
371         # log_vec gives: omega * axis  (approximately, via Pade)
372         # So: score = (score_mag / sin(omega)) * log_vec ... NO, that's
                wrong.
373         # log_vec = omega * axis, so:
374         #   score = score_mag * (omega / sin omega) * axis
375         #         = score_mag * (1 / sin omega) * (omega * axis)
376         #         = (score_mag / sin omega) * log_vec
377         #
378         # But omega / sin(omega) is already baked into the Pade? No.
379         # The Pade gives: sin(omega) * axis * (omega / sin(omega)) = omega *
                axis.
380         # So log_vec = omega * axis.  Perfect.
381         #
382         # Scale factor: score_mag * (omega / sin omega) / omega
383         #             = score_mag / sin(omega)
384
385         sin_omega = torch.sin(omega_detached).clamp(min=1e-7)
386         # For very small omega, log_vec ~ 0 and score ~ 0, so the scale
387         # doesn't matter much.  Use safe division.
388         omega_safe = omega_detached.clamp(min=1e-7)
389         scale = score_mag * _omega_over_sin_omega(omega_detached) /
                omega_safe
390         # (B, N)
391
```

```
392         # Handle omega ~ 0: when omega < 1e-4, log_vec ~ v ~ sin(omega)*axis
                ~ omega*axis
393         # and score ~ score_mag * 1 * axis = score_mag * axis
394         # scale = score_mag * 1 / omega, and log_vec = omega * axis
395         # => scale * log_vec = score_mag * axis.  Correct.
396
397     # Apply detached scale to differentiable log_vec
398     score_vec = scale.unsqueeze(-1) * log_vec                    # (B, N, 3)
399
400     # NaN safety: replace any NaN elements with zero
401     score_vec = torch.where(torch.isfinite(score_vec), score_vec, torch.
            zeros_like(score_vec))
402
403     return score_vec
404
405
406 #
        ------------------------------------------------------------------------

407 # Legacy rotation loss (Frobenius geodesic, for fallback / comparison)
408 #
        ------------------------------------------------------------------------

409
410 def rotation_loss(
411     r_pred: torch.Tensor,
412     r_true: torch.Tensor,
413     r_noisy: torch.Tensor,
414     t: torch.Tensor,
415     mask: torch.Tensor,
416     igso3_cache=None,
417 ) -> torch.Tensor:
418     """Frobenius geodesic rotation loss (no IGSO3 weighting).
419
420     Kept for backward compatibility and as a fallback.
421     ||log(R_pred^T @ R_true)||^2 averaged over valid residues.
422     """
423     delta = torch.matmul(r_pred.transpose(-2, -1), r_true)
424     log_delta = _log_map_so3(delta)
425     sq = (log_delta ** 2).sum(-1) * mask
426     return sq.sum() / mask.sum().clamp(min=1)
427
428
429 #
        ------------------------------------------------------------------------

430 # Auxiliary backbone-atom reconstruction loss  (applied when t < 0.25)
431 #
        ------------------------------------------------------------------------

432
433 def backbone_atom_loss(
434     r_pred: torch.Tensor,
435     x_pred: torch.Tensor,
436     psi_pred: torch.Tensor,
437     r_true: torch.Tensor,
438     x_true: torch.Tensor,
439     psi_true: torch.Tensor,
440     mask: torch.Tensor,
441 ) -> torch.Tensor:
442     """MSE on reconstructed backbone atoms (N, CA, C, O).
443
```

```
    Only meaningful when t < T_F/4 = 0.25  (caller is responsible for gating).
    """
    atoms_pred = frames_to_atoms(r_pred, x_pred, psi_pred)
    atoms_true = frames_to_atoms(r_true, x_true, psi_true)
    sq = ((atoms_pred - atoms_true) ** 2).sum(-1)
    sq = sq.mean(-1) * mask
    return sq.sum() / mask.sum().clamp(min=1)


#
    ---------------------------------------------------------------------------

# Auxiliary pairwise Calpha distance loss  (applied when t < 0.25)
#
    ---------------------------------------------------------------------------


def pairwise_distance_loss(
    x_pred: torch.Tensor,
    x_true: torch.Tensor,
    mask: torch.Tensor,
    cutoff: float = 0.6,
) -> torch.Tensor:
    """MSE on pairwise Calpha distances for pairs within *cutoff* nm (6 A)."""
    diff_pred = x_pred[:, :, None, :] - x_pred[:, None, :, :]
    d_pred = torch.sqrt((diff_pred ** 2).sum(-1) + 1e-8)

    diff_true = x_true[:, :, None, :] - x_true[:, None, :, :]
    d_true = torch.sqrt((diff_true ** 2).sum(-1) + 1e-8)

    pair_mask = mask[:, :, None] * mask[:, None, :]
    within_cutoff = (d_true < cutoff).float()
    diag_mask = 1.0 - torch.eye(mask.shape[1], device=mask.device, dtype=mask.dtype).unsqueeze(0)
    pair_mask = pair_mask * within_cutoff * diag_mask

    sq = (d_pred - d_true) ** 2
    sq = sq * pair_mask
    return sq.sum() / pair_mask.sum().clamp(min=1)


#
    ---------------------------------------------------------------------------

# Distogram for self-conditioning
#
    ---------------------------------------------------------------------------


def compute_distogram(
    x: torch.Tensor,
    num_bins: int = 32,
    d_min: float = 0.2,
    d_max: float = 2.0,
) -> torch.Tensor:
    """One-hot pairwise distance histogram from Calpha positions.

    Parameters
    ----------
    x     : (B, N, 3)  Calpha positions in nm
    num_bins : int
```

```
    d_min, d_max : float  range in nm  (2-20 A)

    Returns
    -------
    distogram : (B, N, N, num_bins)
    """
    diff = x[:, :, None, :] - x[:, None, :, :]
    dist = torch.sqrt((diff ** 2).sum(-1) + 1e-8)

    bin_edges = torch.linspace(d_min, d_max, num_bins + 1, device=x.device,
        dtype=x.dtype)
    bin_idx = torch.bucketize(dist, bin_edges[1:-1])
    distogram = F.one_hot(bin_idx.long(), num_classes=num_bins).float()
    return distogram
```

Listing 2: Loss Functions (losses_v2.py)

A.3 Training Script (`train_v2.py`)

```python
"""
FrameDiff v2 training script: unconditional protein backbone diffusion.

Usage:
    python train_v2.py [--resume PATH]

Changes from train.py (v1):
- IGSO3 rotation noise (replaces axis-angle placeholder)
- rotation_loss_v2 (proper IGSO3 score-matching)
- IGSO3Cache is REQUIRED (no fallback)
- Gradient monitoring for first epoch
- Default weight directory: weights_v2

Coordinates are in nanometers throughout.
"""

import os, sys, csv, copy, math, time, signal, argparse
from pathlib import Path
from random import random

_LOG = Path("/global/scratch/users/sergiomar10/latentlabs/framediff/slurm/train_v2_debug.log")
def _log(msg):
    line = f"[{time.strftime('%H:%M:%S')}] {msg}"
    print(line, flush=True)
    with open(_LOG, "a") as f: f.write(line + "\n")

_log("Starting imports...")
_log("  stdlib done")

import torch
_log(f"  torch {torch.__version__}, CUDA={torch.cuda.is_available()}")
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils import clip_grad_norm_
from torch.utils.data import DataLoader
_log("  torch modules done")

# ---- project imports ----------------------------------------------------
SCRIPT_DIR = Path(__file__).resolve().parent
sys.path.insert(0, str(SCRIPT_DIR))

_log("  importing data...")
from data import CATHFrameDataset, LengthBucketSampler
_log("  importing losses_v2...")
from losses_v2 import (
    translation_loss,
    rotation_loss_v2,
    backbone_atom_loss,
    pairwise_distance_loss,
    compute_distogram,
)
_log("  importing model_v2...")
from model_v2 import FrameDiffModel
_log("  importing so3_diffusion...")
from so3_diffusion import IGSO3Cache
_log("  all imports done")

```

```
# ----------------------------------------------------------------------------

# Paths
# ----------------------------------------------------------------------------

DATA_JSONL  = "/global/scratch/users/sergiomar10/latentlabs/CATH_ml_takehome/chain_set.jsonl"
SPLITS_JSON = "/global/scratch/users/sergiomar10/latentlabs/CATH_ml_takehome/chain_set_splits.json"
WEIGHT_DIR  = Path("/global/scratch/users/sergiomar10/latentlabs/framediff/weights_v2")
WEIGHT_DIR.mkdir(parents=True, exist_ok=True)

# ----------------------------------------------------------------------------

# Hyperparameters
# ----------------------------------------------------------------------------

MAX_LEN         = 256
BATCH_SIZE      = 8
LR              = 1e-4
WARMUP_STEPS    = 1000     # linear LR warmup
LR_MIN          = 1e-6     # cosine decay floor
MAX_EPOCHS      = 200
GRAD_CLIP       = 1.0
EMA_DECAY       = 0.999
AUX_WEIGHT      = 0.25
AUX_THRESHOLD   = 0.25      # apply auxiliary losses when t < this
T_MIN           = 0.01      # minimum diffusion time
T_MAX           = 1.00
SELF_COND_PROB  = 0.5
PATIENCE        = 999       # effectively disabled -- cosine LR handles convergence
NUM_WORKERS     = 4
DISTOGRAM_BINS = 32

# IGSO3 noise schedule: sigma(t) = sigma_min + t * (sigma_max - sigma_min)
SIGMA_MIN = 0.1
SIGMA_MAX = 1.5


# ----------------------------------------------------------------------------

# Preemption handling  (SLURM SIGUSR1)
# ----------------------------------------------------------------------------

_PREEMPTED = False

def _handle_sigusr1(signum, frame):
    global _PREEMPTED
    print("[train_v2] Received SIGUSR1 -- will checkpoint and exit after this step.")
    _PREEMPTED = True

signal.signal(signal.SIGUSR1, _handle_sigusr1)


```

```
# ----------------------------------------------------------------------------

# EMA helper
# ----------------------------------------------------------------------------


class EMA:
    """Exponential moving average of model parameters."""

    def __init__(self, model: nn.Module, decay: float = 0.999):
        self.decay = decay
        self.shadow = {n: p.data.clone() for n, p in model.named_parameters
            () if p.requires_grad}

    @torch.no_grad()
    def update(self, model: nn.Module):
        for n, p in model.named_parameters():
            if not p.requires_grad:
                continue
            self.shadow[n].mul_(self.decay).add_(p.data, alpha=1.0 - self.
                decay)

    def apply(self, model: nn.Module):
        """Copy EMA weights into model (for evaluation / saving)."""
        for n, p in model.named_parameters():
            if n in self.shadow:
                p.data.copy_(self.shadow[n])

    def state_dict(self):
        return {k: v.clone() for k, v in self.shadow.items()}

    def load_state_dict(self, sd):
        for k, v in sd.items():
            self.shadow[k] = v.clone()


# ----------------------------------------------------------------------------

# Noising utilities
# ----------------------------------------------------------------------------


def center(x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
    """Subtract center of mass (only over valid residues) from positions.

    x    : (B, N, 3)
    mask : (B, N)
    """
    mask_sum = mask.sum(-1, keepdim=True).clamp(min=1)    # (B, 1)
    com = (x * mask[..., None]).sum(1) / mask_sum          # (B, 3)
    return x - com[:, None, :]


def noise_translations(x0: torch.Tensor, t: torch.Tensor,
                       mask: torch.Tensor) -> torch.Tensor:
    """Add Gaussian noise to Calpha translations using OU-style schedule.

    x0 : (B, N, 3)
    t  : (B,)
```

```
    Returns x_t : (B, N, 3)  (centered)
    """
    alpha_t = torch.exp(-t / 2.0)                          # (B,)
    sigma_t = torch.sqrt(1.0 - alpha_t ** 2)               # (B,)
    eps = torch.randn_like(x0)
    x_t = alpha_t[:, None, None] * x0 + sigma_t[:, None, None] * eps
    x_t = center(x_t, mask)
    return x_t


# ---------------------------------------------------------------------------
# Metrics
# ---------------------------------------------------------------------------


def rotation_angle_error(r_pred: torch.Tensor, r_true: torch.Tensor,
                         mask: torch.Tensor) -> float:
    """Average rotation angle error in degrees.

    r_pred, r_true : (B, N, 3, 3)
    mask : (B, N)
    """
    delta = torch.matmul(r_pred.transpose(-2, -1), r_true)
    trace = delta[..., 0, 0] + delta[..., 1, 1] + delta[..., 2, 2]
    cos_theta = ((trace - 1.0) / 2.0).clamp(-1.0, 1.0)
    theta = torch.acos(cos_theta)                          # radians
    theta_deg = theta * 180.0 / math.pi
    return (theta_deg * mask).sum().item() / mask.sum().clamp(min=1).item()


def ca_rmsd(x_pred: torch.Tensor, x_true: torch.Tensor,
            mask: torch.Tensor) -> float:
    """Average Calpha RMSD in Angstroms (input in nm, output in A).

    x_pred, x_true : (B, N, 3)
    mask : (B, N)
    """
    sq = ((x_pred - x_true) ** 2).sum(-1)    # (B, N)  nm^2
    per_sample = (sq * mask).sum(-1) / mask.sum(-1).clamp(min=1)  # (B,) nm
        ^2
    rmsd_nm = torch.sqrt(per_sample).mean().item()
    return rmsd_nm * 10.0   # convert to Angstroms


# ---------------------------------------------------------------------------
# Training helpers
# ---------------------------------------------------------------------------


def _unwrap(model):
    """Unwrap DataParallel if needed."""
    return model.module if hasattr(model, 'module') else model

def save_checkpoint(model, ema, optimizer, epoch, val_loss, path):
    torch.save({
        "epoch": epoch,
        "model_state_dict": _unwrap(model).state_dict(),
```

```
        "ema_state_dict": ema.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
        "val_loss": val_loss,
    }, path)
    print(f"  [checkpoint] saved to {path}")


def load_checkpoint(path, model, ema, optimizer, device):
    ckpt = torch.load(path, map_location=device, weights_only=False)
    _unwrap(model).load_state_dict(ckpt["model_state_dict"])
    ema.load_state_dict(ckpt["ema_state_dict"])
    optimizer.load_state_dict(ckpt["optimizer_state_dict"])
    return ckpt["epoch"], ckpt.get("val_loss", float("inf"))


# ---------------------------------------------------------------------------
# Training
# ---------------------------------------------------------------------------


def train(args):
    global WEIGHT_DIR
    if args.weight_dir:
        WEIGHT_DIR = Path(args.weight_dir)
        WEIGHT_DIR.mkdir(parents=True, exist_ok=True)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"[train_v2] device = {device}")
    print(f"[train_v2] weight_dir = {WEIGHT_DIR}")

    # ---- Data ------------------------------------------------------------
    train_ds = CATHFrameDataset(DATA_JSONL, SPLITS_JSON, split="train",
                                max_len=MAX_LEN, augment=True)
    val_ds   = CATHFrameDataset(DATA_JSONL, SPLITS_JSON, split="validation",
                                max_len=MAX_LEN, augment=False)

    train_sampler = LengthBucketSampler(train_ds, batch_size=BATCH_SIZE,
                                        shuffle=True, drop_last=True)
    train_loader  = DataLoader(train_ds, batch_sampler=train_sampler,
                               num_workers=NUM_WORKERS, pin_memory=True)
    val_loader    = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False,
                               num_workers=NUM_WORKERS, pin_memory=True)

    # ---- Model -----------------------------------------------------------
    model_kwargs = dict(max_len=MAX_LEN, distogram_bins=DISTOGRAM_BINS)
    if args.n_layers is not None:
        model_kwargs["n_layers"] = args.n_layers
    if args.stop_grad:
        model_kwargs["stop_grad_frames"] = True
    model = FrameDiffModel(**model_kwargs)
    print(f"[train_v2] Using FrameDiffModel from model_v2.py (n_layers={
        model.n_layers}, stop_grad={getattr(model, 'stop_grad_frames', False)
        })")

    model = model.to(device)

    # Multi-GPU with DataParallel
    n_gpus = torch.cuda.device_count()
    if n_gpus > 1:
        model = nn.DataParallel(model)
        print(f"[train_v2] DataParallel on {n_gpus} GPUs")
```

```
    else:
        print(f"[train_v2] Single GPU")

    if hasattr(torch, "compile"):
        try:
            # model = torch.compile(model)  # disabled for debugging
            print("[train_v2] torch.compile DISABLED for debugging")
        except Exception:
            pass

    optimizer = torch.optim.Adam(model.parameters(), lr=LR)
    ema = EMA(_unwrap(model), decay=EMA_DECAY)

    # ---- IGSO3 cache (REQUIRED) ----------------------------------------
    _log("  building IGSO3 cache...")
    igso3_cache = IGSO3Cache()
    _log("  IGSO3 cache ready")

    # ---- Resume --------------------------------------------------------
    start_epoch = 0
    best_val_loss = float("inf")
    if args.resume and os.path.isfile(args.resume):
        start_epoch, best_val_loss = load_checkpoint(
            args.resume, model, ema, optimizer, device)
        start_epoch += 1
        print(f"[train_v2] Resumed from epoch {start_epoch}, best_val_loss={
            best_val_loss:.6f}")

    # ---- CSV logger ----------------------------------------------------
    csv_path = WEIGHT_DIR / "train_losses.csv"
    csv_existed = csv_path.exists() and start_epoch > 0
    csv_file = open(csv_path, "a" if csv_existed else "w", newline="")
    csv_writer = csv.writer(csv_file)
    if not csv_existed:
        csv_writer.writerow([
            "epoch", "L_x", "L_r", "L_bb", "L_2d", "total_loss",
            "rot_angle_err_deg", "ca_rmsd_A",
            "val_L_x", "val_L_r", "val_total",
        ])
        csv_file.flush()

    # ---- Training loop -------------------------------------------------
    patience_counter = 0
    self_cond_prev = None
    global_step = 0

    for epoch in range(start_epoch, MAX_EPOCHS):
        model.train()
        epoch_metrics = {
            "L_x": 0.0, "L_r": 0.0, "L_bb": 0.0, "L_2d": 0.0,
            "total": 0.0, "rot_err": 0.0, "rmsd": 0.0, "n_batches": 0,
        }
        t_start = time.time()

        step_in_epoch = 0
        for batch in train_loader:
            step_in_epoch += 1
            r0   = batch["rotations"].to(device)        # (B, N, 3, 3)
            x0   = batch["translations"].to(device)     # (B, N, 3)
            psi  = batch["psi_angles"].to(device)       # (B, N)
            mask = batch["mask"].to(device)              # (B, N)
            B, N = x0.shape[:2]

```

```
            # Sample timestep
            t = torch.rand(B, device=device) * (T_MAX - T_MIN) + T_MIN

            # Noise translations
            x_t = noise_translations(x0, t, mask)

            # Noise rotations using IGSO3
            # sigma(t) = sigma_min + t * (sigma_max - sigma_min)
            sigma = SIGMA_MIN + t * (SIGMA_MAX - SIGMA_MIN)  # (B,)
            t_igso3 = sigma ** 2  # parameter for IGSO3 density: exp(-l(l+1)
                *t/2)

            # Sample rotation noise per-residue
            r_noise = torch.zeros(B, N, 3, 3, device=device, dtype=x0.dtype)
            for b in range(B):
                r_noise_b = igso3_cache.sample(t_igso3[b].item(), N, device=
                    device, dtype=x0.dtype)
                r_noise[b] = r_noise_b  # (N, 3, 3)
            r_t = torch.matmul(r0, r_noise)  # compose clean @ noise

            # Self-conditioning: 50% of steps use previous prediction
            if random() < SELF_COND_PROB and self_cond_prev is not None:
                # Pad/truncate to match current batch dims
                sc = self_cond_prev
                if sc.shape[0] != B or sc.shape[1] != N:
                    distogram = torch.zeros(B, N, N, DISTOGRAM_BINS,
                                            device=device)
                else:
                    distogram = compute_distogram(sc)
            else:
                distogram = torch.zeros(B, N, N, DISTOGRAM_BINS, device=
                    device)

            # Forward pass
            optimizer.zero_grad(set_to_none=True)
            r_pred, x_pred, psi_pred = model(r_t, x_t, t, distogram)

            # Debug: check for NaN in outputs on first batch of first epoch
            if epoch == start_epoch and step_in_epoch < 3:
                _log(f"  DEBUG step {step_in_epoch}: x_pred [{x_pred.min().
                    item():.4f}, {x_pred.max().item():.4f}] "
                     f"r_det {torch.det(r_pred[0,0]).item():.4f} "
                     f"x0 [{x0.min().item():.4f}, {x0.max().item():.4f}]")

            # Losses
            L_x = translation_loss(x_pred, x0, mask)
            L_r = rotation_loss_v2(r_pred, r0, r_t, t, mask, igso3_cache)
            loss = L_x + L_r

            # Auxiliary losses for low noise (t < 0.25)
            t_low_mask = (t < AUX_THRESHOLD).any()
            L_bb_val = 0.0
            L_2d_val = 0.0
            if t_low_mask:
                # Select samples with t < threshold
                low_idx = (t < AUX_THRESHOLD)
                if low_idx.any():
                    L_bb = backbone_atom_loss(
                        r_pred[low_idx], x_pred[low_idx], psi_pred[low_idx],
                        r0[low_idx], x0[low_idx], psi[low_idx],
                        mask[low_idx],
                    )
                    L_2d = pairwise_distance_loss(
```

```
                    x_pred[low_idx], x0[low_idx], mask[low_idx],
                )
                loss = loss + AUX_WEIGHT * (L_bb + L_2d)
                L_bb_val = L_bb.item()
                L_2d_val = L_2d.item()

        if not torch.isfinite(loss):
            _log(f"  [WARN] NaN/Inf loss, skipping (L_x={L_x.item()}, L_r={L_r.item()})")
            optimizer.zero_grad(set_to_none=True)
            continue

        loss.backward()

        # Gradient monitoring for the first epoch
        if epoch == start_epoch and step_in_epoch <= 5:
            for name, p in model.named_parameters():
                if p.grad is not None:
                    if not torch.isfinite(p.grad).all():
                        _log(f"  NaN grad in {name}")
                    grad_n = p.grad.norm().item()
                    if grad_n > 100:
                        _log(f"  Large grad in {name}: {grad_n:.1f}")

        # Check for NaN gradients
        grad_norm = clip_grad_norm_(model.parameters(), GRAD_CLIP)
        if not torch.isfinite(grad_norm):
            # Diagnostic: which parameters have NaN gradients?
            if step_in_epoch <= 3:
                nan_params = []
                for name, p in model.named_parameters():
                    if p.grad is not None and not torch.isfinite(p.grad).all():
                        nan_params.append(name)
                _log(f"  [WARN] NaN gradients in: {nan_params[:10]}")
            else:
                _log(f"  [WARN] NaN gradients, skipping optimizer step")
            optimizer.zero_grad(set_to_none=True)
            continue
        # LR schedule: warmup then cosine decay
        global_step += 1
        total_steps = MAX_EPOCHS * len(train_loader)
        if global_step <= WARMUP_STEPS:
            cur_lr = LR * global_step / WARMUP_STEPS
        else:
            progress = (global_step - WARMUP_STEPS) / max(total_steps - WARMUP_STEPS, 1)
            cur_lr = LR_MIN + 0.5 * (LR - LR_MIN) * (1.0 + math.cos(math.pi * progress))
        for pg in optimizer.param_groups:
            pg['lr'] = cur_lr

        optimizer.step()
        ema.update(_unwrap(model))

        # Update self-conditioning cache
        with torch.no_grad():
            self_cond_prev = x_pred.detach()

        # Metrics
        with torch.no_grad():
            rot_err = rotation_angle_error(r_pred, r0, mask)
            rmsd = ca_rmsd(x_pred, x0, mask)
```

```
448
449             epoch_metrics["L_x"]       += L_x.item()
450             epoch_metrics["L_r"]       += L_r.item()
451             epoch_metrics["L_bb"]      += L_bb_val
452             epoch_metrics["L_2d"]      += L_2d_val
453             epoch_metrics["total"]     += loss.item()
454             epoch_metrics["rot_err"]   += rot_err
455             epoch_metrics["rmsd"]      += rmsd
456             epoch_metrics["n_batches"] += 1
457
458             if _PREEMPTED:
459                 break
460
461         nb = max(epoch_metrics["n_batches"], 1)
462         elapsed = time.time() - t_start
463
464         # ---- Validation ------------------------------------------------
465         model.eval()
466         val_metrics = {"L_x": 0.0, "L_r": 0.0, "total": 0.0, "n": 0}
467         with torch.no_grad():
468             for batch in val_loader:
469                 r0   = batch["rotations"].to(device)
470                 x0   = batch["translations"].to(device)
471                 psi  = batch["psi_angles"].to(device)
472                 mask = batch["mask"].to(device)
473                 B, N = x0.shape[:2]
474
475                 t = torch.rand(B, device=device) * (T_MAX - T_MIN) + T_MIN
476                 x_t = noise_translations(x0, t, mask)
477
478                 # IGSO3 rotation noise for validation
479                 sigma = SIGMA_MIN + t * (SIGMA_MAX - SIGMA_MIN)
480                 t_igso3 = sigma ** 2
481                 r_noise = torch.zeros(B, N, 3, 3, device=device, dtype=x0.
                        dtype)
482                 for b in range(B):
483                     r_noise_b = igso3_cache.sample(t_igso3[b].item(), N,
                            device=device, dtype=x0.dtype)
484                     r_noise[b] = r_noise_b
485                 r_t = torch.matmul(r0, r_noise)
486
487                 distogram = torch.zeros(B, N, N, DISTOGRAM_BINS, device=
                        device)
488
489                 r_pred, x_pred, psi_pred = model(r_t, x_t, t, distogram)
490                 vL_x = translation_loss(x_pred, x0, mask)
491                 vL_r = rotation_loss_v2(r_pred, r0, r_t, t, mask,
                        igso3_cache)
492                 val_metrics["L_x"]   += vL_x.item()
493                 val_metrics["L_r"]   += vL_r.item()
494                 val_metrics["total"] += (vL_x + vL_r).item()
495                 val_metrics["n"]     += 1
496
497         vn = max(val_metrics["n"], 1)
498         val_total = val_metrics["total"] / vn
499
500         # ---- Logging ---------------------------------------------------
501         print(
502             f"Epoch {epoch:03d} ({elapsed:.0f}s)  "
503             f"L_x={epoch_metrics['L_x']/nb:.5f}  "
504             f"L_r={epoch_metrics['L_r']/nb:.5f}  "
505             f"L_bb={epoch_metrics['L_bb']/nb:.5f}  "
506             f"L_2d={epoch_metrics['L_2d']/nb:.5f}  "
```

```
            f"total={epoch_metrics['total']/nb:.5f}  "
            f"rot_err={epoch_metrics['rot_err']/nb:.1f}deg  "
            f"RMSD={epoch_metrics['rmsd']/nb:.2f}A  "
            f"| val_L_x={val_metrics['L_x']/vn:.5f}  "
            f"val_L_r={val_metrics['L_r']/vn:.5f}  "
            f"val_total={val_total:.5f}"
        )

        csv_writer.writerow([
            epoch,
            f"{epoch_metrics['L_x']/nb:.6f}",
            f"{epoch_metrics['L_r']/nb:.6f}",
            f"{epoch_metrics['L_bb']/nb:.6f}",
            f"{epoch_metrics['L_2d']/nb:.6f}",
            f"{epoch_metrics['total']/nb:.6f}",
            f"{epoch_metrics['rot_err']/nb:.2f}",
            f"{epoch_metrics['rmsd']/nb:.3f}",
            f"{val_metrics['L_x']/vn:.6f}",
            f"{val_metrics['L_r']/vn:.6f}",
            f"{val_total:.6f}",
        ])
        csv_file.flush()

        # ---- Checkpointing ------------------------------------------------
        is_best = val_total < best_val_loss
        if is_best:
            best_val_loss = val_total
            patience_counter = 0
            save_checkpoint(model, ema, optimizer, epoch, val_total,
                            WEIGHT_DIR / "best.pt")
        else:
            patience_counter += 1

        # Always save latest
        save_checkpoint(model, ema, optimizer, epoch, val_total,
                        WEIGHT_DIR / "latest.pt")

        # ---- Early stopping / preemption ----------------------------------
        if patience_counter >= PATIENCE:
            print(f"[train_v2] Early stopping at epoch {epoch} "
                  f"(no improvement for {PATIENCE} epochs)")
            break

        if _PREEMPTED:
            print("[train_v2] Preempted -- checkpoint saved, exiting.")
            # Resubmit SLURM job
            job_script = str(SCRIPT_DIR.parent / "slurm" / "train_v2.sh")
            if os.path.isfile(job_script):
                os.system(f"sbatch {job_script}")
                print(f"[train_v2] Resubmitted {job_script}")
            break

    csv_file.close()
    print(f"[train_v2] Done.  Best val loss = {best_val_loss:.6f}")
    print(f"[train_v2] Logs    -> {csv_path}")
    print(f"[train_v2] Weights -> {WEIGHT_DIR}")


# ---------------------------------------------------------------------------

# Entry point
```

```
# ----------------------------------------------------------------------------

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="FrameDiff v2 training")
    parser.add_argument("--resume", type=str, default=None,
                        help="Path to checkpoint to resume from")
    parser.add_argument("--n_layers", type=int, default=None,
                        help="Number of model layers (default: model default)")
    parser.add_argument("--weight_dir", type=str, default=None,
                        help="Override weight directory")
    parser.add_argument("--stop_grad", action="store_true",
                        help="Enable stop-gradient on intermediate frames")
    args = parser.parse_args()
    train(args)
```

Listing 3: Training Script (train_v2.py)

A.4 Sampling Script (sample_v2.py)

```python
"""
FrameDiff v2 sampling: generate protein backbones via reverse SE(3) diffusion.

Changes from sample.py (v1):
- Geodesic reverse SDE on SO(3) (replaces naive matrix interpolation)
- IGSO3 for initial rotation noise (replaces QR decomposition)
- Proper timestep-to-sigma schedule
- Import from model_v2 and so3_diffusion

Usage:
    python sample_v2.py --weights PATH --n_samples 10 --length 100 --out_dir ./generated
"""

import os, sys, argparse, math, time
from pathlib import Path

import torch
import torch.nn.functional as F
import numpy as np

SCRIPT_DIR = Path(__file__).resolve().parent
sys.path.insert(0, str(SCRIPT_DIR))

from model_v2 import FrameDiffModel, compute_distogram
from losses import frames_to_atoms
from so3_diffusion import (IGSO3Cache, axis_angle_to_matrix, matrix_to_axis_angle,
                           so3_exp, alpha_t, sigma_t)


# ---------------------------------------------------------------------------
# IGSO3 noise schedule constants
# ---------------------------------------------------------------------------
SIGMA_MIN = 0.1
SIGMA_MAX = 1.5


# ---------------------------------------------------------------------------
# SO(3) reverse diffusion step
# ---------------------------------------------------------------------------

def reverse_step_rotations(R_t, R_pred, sigma_now, sigma_next, igso3_cache):
    """One reverse diffusion step on SO(3) using geodesic interpolation.

    Parameters
    ----------
    R_t        : (B_N, 3, 3)  current noisy rotations (flattened batch*N)
    R_pred     : (B_N, 3, 3)  model-predicted clean rotations
    sigma_now  : float, current noise level
    sigma_next : float, next noise level (smaller)
```

```
    igso3_cache : IGSO3Cache instance

    Returns
    -------
    R_new : (B_N, 3, 3) updated rotations
    """
    B_N = R_t.shape[0]

    # Relative rotation: how far is R_t from R_pred?
    R_rel = R_t.transpose(-2, -1) @ R_pred  # rotation from R_t to R_pred
    axis, omega = matrix_to_axis_angle(R_rel)  # axis (B_N, 3), omega (B_N,)

    # How much to move toward R_pred at this step
    # Posterior scaling: move proportionally to noise reduction
    if sigma_now > 1e-6:
        frac = 1.0 - (sigma_next / sigma_now) ** 2  # fraction of noise to
            remove
    else:
        frac = 1.0
    omega_step = omega * frac

    # Apply geodesic step toward R_pred
    R_step = axis_angle_to_matrix(axis, omega_step)
    R_mean = R_t @ R_step

    # Add reverse-step noise (IGSO3)
    if sigma_next > 0.01:  # don't add noise at final step
        noise_sigma = sigma_next * math.sqrt(1.0 - (sigma_next / max(
            sigma_now, 1e-6)) ** 2)
        t_noise = noise_sigma ** 2
        t_noise = max(t_noise, 0.001)  # clamp for cache range
        R_noise = igso3_cache.sample(t_noise, B_N, device=R_t.device, dtype=
            R_t.dtype)
        R_new = R_mean @ R_noise
    else:
        R_new = R_mean

    # Gram-Schmidt re-orthogonalize
    c0 = R_new[..., :, 0]
    c1 = R_new[..., :, 1]
    e0 = F.normalize(c0, dim=-1)
    e1 = c1 - (c1 * e0).sum(-1, keepdim=True) * e0
    e1 = F.normalize(e1, dim=-1)
    e2 = torch.cross(e0, e1, dim=-1)
    R_new = torch.stack([e0, e1, e2], dim=-1)

    return R_new


#
    ----------------------------------------------------------------------------

# Reverse diffusion sampler
#
    ----------------------------------------------------------------------------


@torch.no_grad()
def sample_backbones(
    model,
    igso3_cache,
    n_samples: int = 10,
    length: int = 100,
```

```
    n_steps: int = 200,
    device: torch.device = torch.device("cpu"),
    self_condition: bool = True,
):
    """Generate protein backbones via reverse SE(3) diffusion.

    Uses DDPM-style x0-prediction with proper IGSO3 rotation diffusion.

    Returns
    -------
    all_atoms : (n_samples, length, 4, 3) backbone atoms [N, CA, C, O] in nm
    all_R     : (n_samples, length, 3, 3) final rotations
    all_x     : (n_samples, length, 3) final Ca positions
    all_psi   : (n_samples, length) final psi angles
    """
    model.eval()
    dtype = next(model.parameters()).dtype

    # Timestep schedule: T -> 0
    timesteps = torch.linspace(1.0, 0.0, n_steps + 1)
    sigmas = SIGMA_MIN + timesteps * (SIGMA_MAX - SIGMA_MIN)

    # Initialize from noise
    # Translations: isotropic Gaussian, centered
    x_t = torch.randn(n_samples, length, 3, device=device, dtype=dtype)
    x_t = x_t - x_t.mean(dim=1, keepdim=True)  # center

    # Rotations: IGSO3 at maximum noise
    t_max_igso3 = SIGMA_MAX ** 2
    R_t = igso3_cache.sample(t_max_igso3, n_samples * length, device=device,
        dtype=dtype)
    R_t = R_t.view(n_samples, length, 3, 3)

    # Self-conditioning cache
    distogram = torch.zeros(n_samples, length, length, 32, device=device,
        dtype=dtype)

    print(f"Sampling {n_samples} backbones of length {length} with {n_steps}
        steps...")

    for i in range(n_steps):
        t_now = timesteps[i]
        t_next = timesteps[i + 1]
        sigma_now = sigmas[i].item()
        sigma_next = sigmas[i + 1].item()
        t_batch = t_now.expand(n_samples).to(device)

        # Model prediction: clean frames
        R_pred, x_pred, psi_pred = model(R_t, x_t, t_batch, distogram)

        # Self-conditioning: update distogram from prediction
        if self_condition:
            distogram = compute_distogram(x_pred)

        # --- Translation reverse step ---
        # VP-SDE: x_t = alpha_t * x0 + sigma_t * eps
        alpha_now = torch.exp(-t_now / 2.0)
        sigma_now_trans = torch.sqrt(1.0 - alpha_now ** 2)
        alpha_next_val = torch.exp(-t_next / 2.0)
        sigma_next_trans = torch.sqrt(1.0 - alpha_next_val ** 2)

        if i < n_steps - 1:  # don't add noise on last step
            # DDPM posterior mean using x0 prediction
```

```
            noise = torch.randn_like(x_t)
            noise = noise - noise.mean(dim=1, keepdim=True)  # center
            x_t = alpha_next_val * x_pred + sigma_next_trans * noise
        else:
            x_t = x_pred

        x_t = x_t - x_t.mean(dim=1, keepdim=True)  # re-center

        # --- Rotation reverse step (geodesic on SO(3)) ---
        # Flatten to (n_samples*length, 3, 3) for the reverse step
        R_t_flat = R_t.view(-1, 3, 3)
        R_pred_flat = R_pred.view(-1, 3, 3)
        R_t_flat = reverse_step_rotations(R_t_flat, R_pred_flat,
                                          sigma_now, sigma_next,
                                             igso3_cache)
        R_t = R_t_flat.view(n_samples, length, 3, 3)

        if (i + 1) % 50 == 0 or i == n_steps - 1:
            print(f"  Step {i+1}/{n_steps}: t={t_next.item():.4f}, "
                  f"sigma={sigma_next:.4f}")

    # Reconstruct full backbone atoms
    atoms = frames_to_atoms(R_t, x_t, psi_pred)  # (n_samples, length, 4, 3)
    return atoms, R_t, x_t, psi_pred


#
    ------------------------------------------------------------------------

# PDB writer
#
    ------------------------------------------------------------------------


def atoms_to_pdb(atoms: np.ndarray, path: str, chain: str = "A"):
    """Write backbone atoms (N, CA, C, O) to PDB file.

    atoms : (L, 4, 3) in nanometers
    """
    atom_names = ["N", "CA", "C", "O"]
    elements = ["N", "C", "C", "O"]

    with open(path, "w") as f:
        atom_idx = 1
        for res_idx in range(atoms.shape[0]):
            for a_idx, (aname, elem) in enumerate(zip(atom_names, elements))
                :
                x, y, z = atoms[res_idx, a_idx] * 10.0  # nm -> Angstroms
                f.write(
                    f"ATOM  {atom_idx:5d}  {aname:<3s} GLY {chain}{res_idx
                        +1:4d}    "
                    f"{x:8.3f}{y:8.3f}{z:8.3f}  1.00  0.00           {elem
                        :>2s}\n"
                )
                atom_idx += 1
        f.write("END\n")


#
    ------------------------------------------------------------------------

# Main
```

```python
# ----------------------------------------------------------------------------


def main():
    parser = argparse.ArgumentParser(description="FrameDiff v2 backbone sampling")
    parser.add_argument("--weights", type=str,
                        default="/global/scratch/users/sergiomar10/latentlabs/framediff/weights_v2/best.pt",
                        help="Path to model checkpoint")
    parser.add_argument("--n_samples", type=int, default=8, help="Number of samples")
    parser.add_argument("--length", type=int, default=100, help="Chain length")
    parser.add_argument("--n_steps", type=int, default=200, help="Diffusion steps")
    parser.add_argument("--out_dir", type=str,
                        default="/global/scratch/users/sergiomar10/latentlabs/framediff/generated_v2",
                        help="Output directory")
    parser.add_argument("--device", type=str, default="cuda" if torch.cuda.is_available() else "cpu")
    args = parser.parse_args()

    device = torch.device(args.device)
    out_dir = Path(args.out_dir)
    out_dir.mkdir(parents=True, exist_ok=True)

    # Load model
    print(f"Loading model from {args.weights}")
    model = FrameDiffModel(max_len=512, distogram_bins=32).to(device)
    ckpt = torch.load(args.weights, map_location=device, weights_only=False)

    # Load EMA weights if available
    if "ema_state_dict" in ckpt:
        print("Using EMA weights")
        ema_sd = ckpt["ema_state_dict"]
        model_sd = model.state_dict()
        for k in model_sd:
            if k in ema_sd:
                model_sd[k] = ema_sd[k]
        model.load_state_dict(model_sd)
    else:
        model.load_state_dict(ckpt["model_state_dict"])

    epoch = ckpt.get("epoch", "?")
    val_loss = ckpt.get("val_loss", "?")
    print(f"Loaded checkpoint: epoch={epoch}, val_loss={val_loss}")

    # Build IGSO3 cache
    print("Building IGSO3 cache...")
    igso3_cache = IGSO3Cache()
    print("IGSO3 cache ready")

    # Sample
    t0 = time.time()
    atoms, R, x, psi = sample_backbones(
        model,
        igso3_cache,
        n_samples=args.n_samples,
        length=args.length,
        n_steps=args.n_steps,
```

```
        device=device,
    )
    elapsed = time.time() - t0
    print(f"Sampling took {elapsed:.1f}s")

    # Save PDBs
    atoms_np = atoms.cpu().numpy()
    for i in range(args.n_samples):
        pdb_path = out_dir / f"sample_{i:03d}_L{args.length}.pdb"
        atoms_to_pdb(atoms_np[i], str(pdb_path))
        print(f"  Saved {pdb_path}")

    # Basic quality stats
    print("\n--- Quick quality check ---")
    for i in range(min(3, args.n_samples)):
        ca = atoms_np[i, :, 1, :] * 10.0  # CA positions in Angstroms
        # Ca-Ca bond lengths
        dists = np.sqrt(((ca[1:] - ca[:-1]) ** 2).sum(-1))
        print(f"  Sample {i}: Ca-Ca distances: mean={dists.mean():.2f}A, "
              f"std={dists.std():.2f}A, min={dists.min():.2f}A, max={dists.
                  max():.2f}A "
              f"(ideal: 3.8A)")
        # Radius of gyration
        com = ca.mean(axis=0)
        rg = np.sqrt(((ca - com) ** 2).sum(-1).mean())
        print(f"           Rg={rg:.1f}A")

    print(f"\nAll {args.n_samples} PDBs saved to {out_dir}")


if __name__ == "__main__":
    main()
```

Listing 4: Sampling Script (sample_v2.py)

A.5 SO(3) Diffusion (so3_diffusion.py)

```python
"""
SO(3) rotation diffusion for FrameDiff-style protein backbone generation.

Implements the Isotropic Gaussian on SO(3) (IGSO3) distribution, its score
function, sampling via CDF inversion, and the associated rotation utilities
(Rodrigues, exp/log maps).  Also includes VP-SDE translation diffusion.

All public functions operate on PyTorch tensors.  The IGSO3Cache class
pre-computes grids with numpy/scipy and exposes fast torch-based lookups.

Reference
---------
Yim et al., "SE(3) Diffusion Model with Application to Protein Backbone
Generation", ICML 2023.
"""

from __future__ import annotations

import ast
import math
from typing import Optional, Tuple

import numpy as np
import torch
from scipy.interpolate import RegularGridInterpolator
from scipy.integrate import trapezoid

# ---------------------------------------------------------------------------
# Numerical helpers
# ---------------------------------------------------------------------------

_EPS = 1e-7  # guard for division-by-zero


def _sinc_half(omega: np.ndarray) -> np.ndarray:
    """sin(omega/2) / (omega/2), safe at omega=0."""
    h = omega / 2.0
    out = np.ones_like(h)
    mask = np.abs(h) > _EPS
    out[mask] = np.sin(h[mask]) / h[mask]
    return out


# ---------------------------------------------------------------------------
# 1. IGSO3 Density
# ---------------------------------------------------------------------------


def _igso3_density_np(
    omega: np.ndarray,
    t: np.ndarray,
    L_max: int = 1000,
) -> np.ndarray:
```

```
    """
    Evaluate the IGSO3 density on a meshgrid (omega, t) using numpy float64.

    f(omega, t) = sum_{l=0}^{L_max} (2l+1) exp(-l(l+1) t/2)
                  * sin((l+0.5) omega) / sin(omega/2)

    Parameters
    ----------
    omega : (N,) array, angles in [0, pi].
    t     : (M,) array, diffusion times > 0.

    Returns
    -------
    density : (N, M) array.
    """
    omega = np.asarray(omega, dtype=np.float64)
    t = np.asarray(t, dtype=np.float64)

    # Shape: (N, 1) and (1, M) for broadcasting
    om = omega[:, None]
    tt = t[None, :]

    density = np.zeros((omega.shape[0], t.shape[0]), dtype=np.float64)

    sin_half = np.sin(om / 2.0)  # (N,1)
    # Near omega=0 or omega=pi, sin(omega/2) -> 0; handle via Taylor.
    safe_denom = np.where(np.abs(sin_half) > _EPS, sin_half, np.ones_like(
        sin_half))

    for ell in range(L_max + 1):
        coeff = (2 * ell + 1)
        exp_part = np.exp(-ell * (ell + 1) * tt / 2.0)  # (1, M)

        # sin((ell+0.5)*omega) with Taylor near omega=0
        arg = (ell + 0.5) * om  # (N,1)
        sin_num = np.sin(arg)

        # ratio = sin((l+0.5)*omega) / sin(omega/2)
        # At omega->0: Taylor gives (2l+1) (leading order)
        # At omega->pi: sin(omega/2)->1, so no singularity in denominator,
        #   but sin((l+0.5)*pi) = cos(l*pi)*sin(pi/2)*... we just compute
            directly
        #   since sin(pi/2)=1 is fine.
        near_zero = np.abs(om) < _EPS
        ratio = np.where(
            near_zero,
            (2 * ell + 1) * np.ones_like(om),  # L'Hopital: limit is (2l+1)
            sin_num / safe_denom,
        )

        density += coeff * exp_part * ratio  # (N, M)

        # Early termination: if the exponential is negligibly small for all
            t
        # remaining terms are even smaller because ell grows.
        if ell > 0 and np.all(exp_part < 1e-30):
            break

    return density


def igso3_density(
    omega: torch.Tensor,
```

```
    t: torch.Tensor,
    L_max: int = 1000,
) -> torch.Tensor:
    """
    IGSO3 density f(omega, t).

    Parameters
    ----------
    omega : (*,) tensor of rotation angles in [0, pi].
    t     : (*,) tensor of diffusion times (same shape or broadcastable).

    Returns
    -------
    density : (*,) tensor.
    """
    shape = torch.broadcast_shapes(omega.shape, t.shape)
    om_flat = omega.broadcast_to(shape).reshape(-1).double().cpu().numpy()
    t_flat = t.broadcast_to(shape).reshape(-1).double().cpu().numpy()

    # Evaluate one-by-one through vectorised 1D call (each pair
        independently)
    unique_t = np.unique(t_flat)
    density_out = np.empty_like(om_flat)

    # Group by unique t for efficiency
    for tv in unique_t:
        mask = t_flat == tv
        om_sub = om_flat[mask]
        d = _igso3_density_np(om_sub, np.array([tv]), L_max=L_max)
        density_out[mask] = d[:, 0]

    return torch.tensor(density_out, dtype=omega.dtype, device=omega.device)
        .reshape(shape)


#
    ----------------------------------------------------------------------------

# 2. IGSO3 Score  (d log f / d omega)
#
    ----------------------------------------------------------------------------


def _igso3_density_and_deriv_np(
    omega: np.ndarray,
    t: np.ndarray,
    L_max: int = 1000,
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Compute f(omega,t) and df/domega on a meshgrid.

    The derivative of the l-th term w.r.t. omega:
        d/domega [ sin((l+0.5)*omega) / sin(omega/2) ]
        = [(l+0.5)*cos((l+0.5)*omega)*sin(omega/2)
           - sin((l+0.5)*omega)*0.5*cos(omega/2)] / sin^2(omega/2)

    Returns (N,M) arrays for density and its omega-derivative.
    """
    omega = np.asarray(omega, dtype=np.float64)
    t = np.asarray(t, dtype=np.float64)
    om = omega[:, None]
    tt = t[None, :]

```

```
    density = np.zeros((len(omega), len(t)), dtype=np.float64)
    deriv = np.zeros_like(density)

    sin_half = np.sin(om / 2.0)
    cos_half = np.cos(om / 2.0)
    sin_half_sq = sin_half ** 2
    near_zero = np.abs(om) < _EPS
    # Near pi: sin(omega/2) ~ 1, cos(omega/2) ~ 0, well-behaved.
    safe_sin_half = np.where(np.abs(sin_half) > _EPS, sin_half, np.ones_like
        (sin_half))
    safe_sin_half_sq = np.where(np.abs(sin_half_sq) > _EPS, sin_half_sq, np.
        ones_like(sin_half_sq))

    for ell in range(L_max + 1):
        coeff = 2 * ell + 1
        half_ell = ell + 0.5
        exp_part = np.exp(-ell * (ell + 1) * tt / 2.0)

        sin_num = np.sin(half_ell * om)
        cos_num = np.cos(half_ell * om)

        # --- f term ---
        ratio = np.where(near_zero, float(coeff), sin_num / safe_sin_half)
        density += coeff * exp_part * ratio

        # --- df/domega term ---
        # d/domega [sin(a*om)/sin(om/2)]
        #   = [a*cos(a*om)*sin(om/2) - sin(a*om)*cos(om/2)/2] / sin^2(om/2)
        # At omega->0 Taylor:
        #   ratio ~ (2l+1) + higher order => derivative at 0:
        #   Expanding to first order: deriv ~ -(2l+1)(l)(l+1)/6 * omega  (->
            0)
        numer = half_ell * cos_num * sin_half - sin_num * cos_half * 0.5
        d_ratio = np.where(near_zero, 0.0, numer / safe_sin_half_sq)
        deriv += coeff * exp_part * d_ratio

        if ell > 0 and np.all(exp_part < 1e-30):
            break

    return density, deriv


def igso3_score(
    omega: torch.Tensor,
    t: torch.Tensor,
    L_max: int = 1000,
) -> torch.Tensor:
    """
    IGSO3 score: d log f / d omega.

    Parameters
    ----------
    omega : (*,) rotation angles in [0, pi].
    t     : (*,) diffusion times.

    Returns
    -------
    score : (*,) tensor, same shape.
    """
    shape = torch.broadcast_shapes(omega.shape, t.shape)
    om_flat = omega.broadcast_to(shape).reshape(-1).double().cpu().numpy()
    t_flat = t.broadcast_to(shape).reshape(-1).double().cpu().numpy()

```

```
    score_out = np.empty_like(om_flat)
    for tv in np.unique(t_flat):
        mask = t_flat == tv
        om_sub = om_flat[mask]
        f, df = _igso3_density_and_deriv_np(om_sub, np.array([tv]), L_max=
            L_max)
        safe_f = np.where(np.abs(f[:, 0]) > 1e-30, f[:, 0], 1e-30)
        score_out[mask] = df[:, 0] / safe_f

    return torch.tensor(score_out, dtype=omega.dtype, device=omega.device).
        reshape(shape)


#
    ----------------------------------------------------------------------------

# 3. Score Normalization Factor
#
    ----------------------------------------------------------------------------


def score_norm(
    t: float | np.ndarray,
    n_omega: int = 2000,
    L_max: int = 1000,
) -> np.ndarray:
    r"""
    Compute 1 / E[||score||^2] for loss weighting.

    E[||score||^2] = \int_0^pi (d log f / d omega)^2 f(omega,t) sin^2(omega
        /2) d omega
                     / \int_0^pi f(omega,t) sin^2(omega/2) d omega

    Parameters
    ----------
    t : scalar or 1-D array of diffusion times.

    Returns
    -------
    lam : same shape as t, the normalisation lambda_r(t).
    """
    t = np.atleast_1d(np.asarray(t, dtype=np.float64))
    omega_grid = np.linspace(1e-6, np.pi - 1e-6, n_omega)

    f, df = _igso3_density_and_deriv_np(omega_grid, t, L_max=L_max)  # (N, M
        )
    sin2_half = (np.sin(omega_grid / 2.0) ** 2)[:, None]  # (N, 1)

    # d log f / d omega
    safe_f = np.where(np.abs(f) > 1e-30, f, 1e-30)
    score_vals = df / safe_f  # (N, M)

    integrand_num = score_vals ** 2 * f * sin2_half
    integrand_den = f * sin2_half

    e_score_sq = trapezoid(integrand_num, omega_grid, axis=0) / \
                 np.maximum(trapezoid(integrand_den, omega_grid, axis=0), 1e
                     -30)

    lam = 1.0 / np.maximum(e_score_sq, 1e-30)
    return lam.squeeze()


```

```python
# ----------------------------------------------------------------------------

# 5. Rodrigues Formula  &  axis-angle <-> matrix
# ----------------------------------------------------------------------------


def _skew_symmetric(v: torch.Tensor) -> torch.Tensor:
    """
    Skew-symmetric matrix from vector.  v: (..., 3) -> (..., 3, 3).
    """
    z = torch.zeros_like(v[..., 0])
    K = torch.stack([
        torch.stack([z, -v[..., 2], v[..., 1]], dim=-1),
        torch.stack([v[..., 2], z, -v[..., 0]], dim=-1),
        torch.stack([-v[..., 1], v[..., 0], z], dim=-1),
    ], dim=-2)
    return K


def axis_angle_to_matrix(axis: torch.Tensor, angle: torch.Tensor) -> torch.Tensor:
    """
    Rodrigues formula.

    Parameters
    ----------
    axis  : (..., 3) unit vectors.
    angle : (...,) rotation angles in radians.

    Returns
    -------
    R : (..., 3, 3) rotation matrices.
    """
    K = _skew_symmetric(axis)  # (..., 3, 3)
    s = torch.sin(angle)[..., None, None]
    c = torch.cos(angle)[..., None, None]
    I = torch.eye(3, dtype=axis.dtype, device=axis.device).expand_as(K)
    R = I + s * K + (1.0 - c) * (K @ K)
    return R


def matrix_to_axis_angle(R: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Extract axis and angle from a rotation matrix.

    Parameters
    ----------
    R : (..., 3, 3) rotation matrices.

    Returns
    -------
    axis  : (..., 3) unit axis  (arbitrary when angle ~ 0).
    angle : (...,)   rotation angle in [0, pi].
    """
    # angle from trace
    trace = R[..., 0, 0] + R[..., 1, 1] + R[..., 2, 2]
    cos_angle = (trace - 1.0) / 2.0
    cos_angle = cos_angle.clamp(-1.0, 1.0)
    angle = torch.acos(cos_angle)  # [0, pi]

```

```
    # axis from skew part of R
    rx = R[..., 2, 1] - R[..., 1, 2]
    ry = R[..., 0, 2] - R[..., 2, 0]
    rz = R[..., 1, 0] - R[..., 0, 1]
    axis = torch.stack([rx, ry, rz], dim=-1)

    # normalise; when angle~0 the axis is arbitrary
    norm = axis.norm(dim=-1, keepdim=True).clamp(min=_EPS)
    axis = axis / norm

    # When angle ~ pi, the skew part is near zero; recover axis from
    # the symmetric part:  R + I = 2 * cos(0) * outer(axis, axis)  (not needed
    # for typical diffusion angles, but handle for robustness).
    near_pi = (angle > math.pi - 0.01).unsqueeze(-1)
    if near_pi.any():
        # Pick the column of (R + I) with the largest diagonal entry
        Rpi = R + torch.eye(3, dtype=R.dtype, device=R.device)
        diag = torch.diagonal(Rpi, dim1=-2, dim2=-1)  # (..., 3)
        idx = diag.argmax(dim=-1)  # (...,)
        # gather the column
        idx_exp = idx.unsqueeze(-1).unsqueeze(-1).expand(*idx.shape, 3, 1)
        col = torch.gather(Rpi, -1, idx_exp).squeeze(-1)  # (..., 3)
        col = col / col.norm(dim=-1, keepdim=True).clamp(min=_EPS)
        axis = torch.where(near_pi, col, axis)

    return axis, angle


#
    ---------------------------------------------------------------------------
# 6. SO(3) Exponential and Log Maps
#
    ---------------------------------------------------------------------------


def so3_exp(w: torch.Tensor) -> torch.Tensor:
    """
    Exponential map: tangent vector (or skew matrix) -> rotation matrix.

    Parameters
    ----------
    w : (..., 3) axis-angle vector  **or**  (..., 3, 3) skew-symmetric matrix.

    Returns
    -------
    R : (..., 3, 3).
    """
    if w.dim() >= 2 and w.shape[-1] == 3 and w.shape[-2] == 3:
        # Extract axis-angle from skew matrix
        w = torch.stack([w[..., 2, 1], w[..., 0, 2], w[..., 1, 0]], dim=-1)

    angle = w.norm(dim=-1).clamp(min=_EPS)
    axis = w / angle.unsqueeze(-1)
    return axis_angle_to_matrix(axis, angle)


def so3_log(R: torch.Tensor) -> torch.Tensor:
    """
    Logarithmic map: rotation matrix -> tangent vector (axis * angle).

```

```
    Parameters
    ----------
    R : (..., 3, 3).

    Returns
    -------
    w : (..., 3) axis-angle vector.
    """
    axis, angle = matrix_to_axis_angle(R)
    return axis * angle.unsqueeze(-1)


#
    ---------------------------------------------------------------------------

# 4. IGSO3 Sampling  (uses Rodrigues from above)
#
    ---------------------------------------------------------------------------


def _sample_axis_uniform(n: int, device: torch.device, dtype: torch.dtype)
    -> torch.Tensor:
    """Sample n unit vectors uniformly on S^2."""
    v = torch.randn(n, 3, device=device, dtype=dtype)
    return v / v.norm(dim=-1, keepdim=True)


def sample_igso3(
    t: float,
    n_samples: int,
    omega_grid: Optional[np.ndarray] = None,
    cdf: Optional[np.ndarray] = None,
    L_max: int = 1000,
    device: torch.device = torch.device("cpu"),
    dtype: torch.dtype = torch.float32,
) -> torch.Tensor:
    """
    Sample rotation matrices from IGSO3(t).

    1. Build / re-use CDF of omega under the IGSO3 measure.
    2. Invert CDF via interpolation to get omega samples.
    3. Sample random axes on S^2.
    4. Construct rotation via Rodrigues.

    Parameters
    ----------
    t         : diffusion time (scalar).
    n_samples : number of rotations to draw.
    omega_grid, cdf : optional pre-computed grid (from IGSO3Cache).
    L_max     : truncation for the density series.

    Returns
    -------
    R : (n_samples, 3, 3) rotation matrices.
    """
    # --- CDF ---
    if omega_grid is None or cdf is None:
        n_grid = 2000
        omega_grid = np.linspace(1e-6, np.pi - 1e-6, n_grid)
        f = _igso3_density_np(omega_grid, np.array([t]), L_max=L_max)[:, 0]
        sin2_half = np.sin(omega_grid / 2.0) ** 2
        pdf = f * sin2_half
        pdf = np.maximum(pdf, 0.0)
```

```
        cdf = np.cumsum(pdf)
        cdf = cdf / cdf[-1]  # normalise

    # Invert CDF by interpolation
    u = np.random.rand(n_samples)
    omega_samples = np.interp(u, cdf, omega_grid)

    omega_t = torch.tensor(omega_samples, dtype=dtype, device=device)
    axes = _sample_axis_uniform(n_samples, device, dtype)
    R = axis_angle_to_matrix(axes, omega_t)
    return R


# ---------------------------------------------------------------------------

# 7. Rotation Score from Predictions
# ---------------------------------------------------------------------------


def compute_rotation_score(
    r_pred: torch.Tensor,
    r_noisy: torch.Tensor,
    t: torch.Tensor,
) -> torch.Tensor:
    """
    Compute the SO(3) score given predicted clean rotation and noised
        rotation.

    score = r_noisy @ (axis * score_magnitude)      (tangent vector at
        r_noisy)

    where  r_rel = r_pred^T @ r_noisy,  omega = angle(r_rel),
           axis  = axis(r_rel),  score_magnitude = d log f / d omega.

    Parameters
    ----------
    r_pred  : (..., 3, 3) predicted clean rotations.
    r_noisy : (..., 3, 3) noised rotations at time t.
    t       : (...,) or scalar diffusion time.

    Returns
    -------
    score : (..., 3) tangent vector at r_noisy (body-frame).
    """
    r_rel = r_pred.transpose(-2, -1) @ r_noisy  # (..., 3, 3)
    axis, omega = matrix_to_axis_angle(r_rel)

    t_b = t.broadcast_to(omega.shape) if isinstance(t, torch.Tensor) else \
        torch.full_like(omega, float(t))
    score_mag = igso3_score(omega, t_b)  # (...,)

    # Tangent vector in body frame at identity, then transport to r_noisy
    tangent_id = axis * score_mag.unsqueeze(-1)  # (..., 3)
    return tangent_id


# ---------------------------------------------------------------------------

# 8. IGSO3 Cache
```

```
# ----------------------------------------------------------------------------


class IGSO3Cache:
    """
    Pre-compute IGSO3 quantities on a 2-D (omega x t) grid and provide fast
    interpolated look-ups.

    Attributes
    ----------
    omega_grid : (N,) numpy array, angles in (0, pi).
    t_grid     : (M,) numpy array, times in (t_min, t_max).
    density    : (N, M) pre-computed f(omega, t).
    deriv      : (N, M) df/domega.
    cdf        : (N, M) CDF(omega | t) for each column.
    score_norms: (M,) lambda_r(t).
    """

    def __init__(
        self,
        n_omega: int = 1000,
        n_t: int = 1000,
        t_min: float = 0.001,
        t_max: float = 5.0,
        L_max: int = 1000,
    ):
        self.L_max = L_max
        self.omega_grid = np.linspace(1e-6, np.pi - 1e-6, n_omega).astype(np
            .float64)
        self.t_grid = np.linspace(t_min, t_max, n_t).astype(np.float64)

        # --- density & derivative ---
        self.density, self.deriv = _igso3_density_and_deriv_np(
            self.omega_grid, self.t_grid, L_max=L_max,
        )

        # --- score = d log f / d omega ---
        safe_f = np.where(np.abs(self.density) > 1e-30, self.density, 1e-30)
        self._score_grid = self.deriv / safe_f   # (N, M)

        # --- CDF for each t column ---
        sin2_half = (np.sin(self.omega_grid / 2.0) ** 2)[:, None]
        pdf = self.density * sin2_half   # (N, M)
        pdf = np.maximum(pdf, 0.0)
        self.cdf = np.cumsum(pdf, axis=0)
        # normalise each column
        cdf_max = self.cdf[-1:, :]
        cdf_max = np.where(cdf_max > 0, cdf_max, 1.0)
        self.cdf = self.cdf / cdf_max

        # --- score norms ---
        self.score_norms = score_norm(self.t_grid, n_omega=n_omega, L_max=
            L_max)

        # --- scipy interpolators ---
        self._density_interp = RegularGridInterpolator(
            (self.omega_grid, self.t_grid), self.density,
            method="linear", bounds_error=False, fill_value=None,
        )
        self._score_interp = RegularGridInterpolator(
            (self.omega_grid, self.t_grid), self._score_grid,
            method="linear", bounds_error=False, fill_value=None,
```

```
        )
        self._score_norm_t = self.t_grid.copy()
        self._score_norm_vals = np.atleast_1d(self.score_norms).copy()

    # ---- fast torch look-ups ----

    def density_lookup(self, omega: torch.Tensor, t: torch.Tensor) -> torch.
        Tensor:
        """Interpolated density f(omega, t)."""
        pts = np.stack([
            omega.double().cpu().numpy().ravel(),
            t.double().cpu().numpy().ravel(),
        ], axis=-1)
        vals = self._density_interp(pts)
        return torch.tensor(vals, dtype=omega.dtype, device=omega.device).
            reshape(omega.shape)

    def score_lookup(self, omega: torch.Tensor, t: torch.Tensor) -> torch.
        Tensor:
        """Interpolated score d log f / d omega."""
        pts = np.stack([
            omega.double().cpu().numpy().ravel(),
            t.double().cpu().numpy().ravel(),
        ], axis=-1)
        vals = self._score_interp(pts)
        return torch.tensor(vals, dtype=omega.dtype, device=omega.device).
            reshape(omega.shape)

    def score_norm_lookup(self, t: torch.Tensor) -> torch.Tensor:
        """Interpolated lambda_r(t)."""
        t_np = t.double().cpu().numpy().ravel()
        vals = np.interp(t_np, self._score_norm_t, self._score_norm_vals)
        return torch.tensor(vals, dtype=t.dtype, device=t.device).reshape(t.
            shape)

    def sample(
        self,
        t: float,
        n_samples: int,
        device: torch.device = torch.device("cpu"),
        dtype: torch.dtype = torch.float32,
    ) -> torch.Tensor:
        """
        Sample rotation matrices from IGSO3(t) using the cached CDF.

        Returns (n_samples, 3, 3).
        """
        # Find the nearest t column for the CDF
        idx = np.searchsorted(self.t_grid, t).clip(0, len(self.t_grid) - 1)
        cdf_col = self.cdf[:, idx]
        return sample_igso3(
            t, n_samples,
            omega_grid=self.omega_grid, cdf=cdf_col,
            L_max=self.L_max, device=device, dtype=dtype,
        )


#
    ------------------------------------------------------------------------

# 9. Translation Diffusion  (VP-SDE, simple Gaussian)
#
    ------------------------------------------------------------------------
```

```python

def alpha_t(t: torch.Tensor) -> torch.Tensor:
    """VP-SDE scaling factor alpha(t) = exp(-t/2)."""
    return torch.exp(-t / 2.0)


def sigma_t(t: torch.Tensor) -> torch.Tensor:
    """VP-SDE noise level sigma(t) = sqrt(1 - alpha(t)^2)."""
    a = alpha_t(t)
    return torch.sqrt((1.0 - a * a).clamp(min=0.0))


def _centering_projection(n: int, device: torch.device, dtype: torch.dtype)
    -> torch.Tensor:
    """
    Centering projection P = I - (1/n) 11^T.  Shape (n, n).
    Removes the mean so translations live in the zero-COM subspace.
    """
    I = torch.eye(n, device=device, dtype=dtype)
    ones = torch.ones(n, 1, device=device, dtype=dtype)
    return I - ones @ ones.T / n


def sample_translation_noise(
    x0: torch.Tensor,
    t: torch.Tensor,
    center: bool = True,
) -> torch.Tensor:
    """
    Forward VP-SDE:  x_t = alpha(t) * x0 + sigma(t) * eps,
    optionally centered (projected to zero COM).

    Parameters
    ----------
    x0 : (N, 3) clean positions.
    t  : scalar or (N,) times.

    Returns
    -------
    x_t : (N, 3).
    """
    if isinstance(t, (int, float)):
        t = torch.tensor(t, dtype=x0.dtype, device=x0.device)
    t = t.reshape(-1, 1) if t.dim() == 1 else t.unsqueeze(0)  # (N,1) or
        (1,1)

    a = alpha_t(t)  # (*, 1)
    s = sigma_t(t)

    eps = torch.randn_like(x0)
    if center:
        P = _centering_projection(x0.shape[0], x0.device, x0.dtype)
        eps = P @ eps  # project noise to zero-COM

    x_t = a * x0 + s * eps
    if center:
        x_t = P @ x_t
    return x_t


def translation_score(
    x0_pred: torch.Tensor,
```

```
    x_t: torch.Tensor,
    t: torch.Tensor,
) -> torch.Tensor:
    """
    Translation score: nabla_{x_t} log p(x_t | x0).

    = -(x_t - alpha(t) x0) / sigma(t)^2

    Parameters
    ----------
    x0_pred : (N, 3) predicted clean positions.
    x_t     : (N, 3) noised positions.
    t       : scalar or (N,) times.

    Returns
    -------
    score : (N, 3).
    """
    if isinstance(t, (int, float)):
        t = torch.tensor(t, dtype=x_t.dtype, device=x_t.device)
    t = t.reshape(-1, 1) if t.dim() == 1 else t.unsqueeze(0)

    a = alpha_t(t)
    s2 = sigma_t(t) ** 2
    s2 = s2.clamp(min=1e-10)
    return -(x_t - a * x0_pred) / s2


# ---------------------------------------------------------------------------
# Self-test / syntax check
# ---------------------------------------------------------------------------

def _run_tests():
    """Quick sanity checks (run with python so3_diffusion.py)."""
    import sys
    print("Running SO(3) diffusion self-tests...")

    # --- Rodrigues roundtrip ---
    axis = torch.tensor([0.0, 0.0, 1.0])
    angle = torch.tensor(1.2)
    R = axis_angle_to_matrix(axis, angle)
    a2, ang2 = matrix_to_axis_angle(R)
    assert abs(ang2.item() - 1.2) < 1e-5, f"Rodrigues roundtrip failed: {ang2}"

    # --- exp / log roundtrip ---
    w = torch.tensor([0.3, -0.5, 0.1])
    R2 = so3_exp(w)
    w2 = so3_log(R2)
    assert torch.allclose(w, w2, atol=1e-5), f"exp/log roundtrip failed"

    # --- Density at moderate t should be peaked near omega=0 ---
    om = torch.tensor([0.01, 0.5, 1.5, 3.0])
    t_mod = torch.tensor([0.5, 0.5, 0.5, 0.5])
    d = igso3_density(om, t_mod)
    assert d[0] > d[2] > d[3], f"Density not peaked at omega=0 for t=0.5: {d}"
    # At very small t (0.01), density is essentially a delta at omega=0
    # so values at omega>=0.5 underflow to ~0, which is correct behaviour.
```

```
741
742     # --- Score sign: should be negative for omega > 0 (pushes back to 0) ---
743     s = igso3_score(torch.tensor([1.0]), torch.tensor([0.5]))
744     assert s.item() < 0, f"Score should be negative, got {s.item()}"
745
746     # --- Sampling: small t → near identity ---
747     R_samp = sample_igso3(0.01, 500)
748     angles = matrix_to_axis_angle(R_samp)[1]
749     assert angles.mean() < 0.5, f"Samples at small t too far from identity: mean angle={angles.mean():.3f}"
750
751     # --- Sampling: large t → ~uniform, mean angle should be ~pi/2 ---
752     R_samp2 = sample_igso3(50.0, 2000)
753     angles2 = matrix_to_axis_angle(R_samp2)[1]
754     # Uniform on SO(3): E[angle] = pi - 1 ≈ 2.14
755     # (density of angle for uniform = sin^2(omega/2) / (pi) normalised)
756     assert abs(angles2.mean().item() - 2.14) < 0.3, \
757         f"Large-t samples not uniform: mean angle={angles2.mean():.3f}, expected ~2.14"
758
759     # --- Score norm should be O(1) ---
760     lam = score_norm(np.array([0.1, 1.0, 3.0]))
761     assert np.all(np.isfinite(lam)), f"Score norm not finite: {lam}"
762     assert np.all(lam > 0), f"Score norm not positive: {lam}"
763
764     # --- Translation VP-SDE ---
765     x0 = torch.randn(10, 3)
766     x0 = x0 - x0.mean(0)
767     xt = sample_translation_noise(x0, 0.5)
768     assert xt.shape == x0.shape
769     assert abs(xt.mean(0).sum().item()) < 0.1, "Translation noise not centered"
770
771     ts = translation_score(x0, xt, 0.5)
772     assert ts.shape == x0.shape
773
774     # --- IGSO3Cache (small grid for speed) ---
775     cache = IGSO3Cache(n_omega=100, n_t=50, L_max=500)
776     d_cached = cache.density_lookup(torch.tensor([1.0]), torch.tensor([0.5]))
777     assert d_cached.shape == (1,)
778     s_cached = cache.score_lookup(torch.tensor([1.0]), torch.tensor([0.5]))
779     assert s_cached.shape == (1,)
780     R_cached = cache.sample(0.5, 10)
781     assert R_cached.shape == (10, 3, 3)
782
783     print("All tests passed.")
784     return 0
785
786
787 if __name__ == "__main__":
788     # Syntax check
789     import pathlib
790     src = pathlib.Path(__file__).read_text()
791     ast.parse(src)
792     _run_tests()
```

Listing 5: SO(3) Diffusion (so3_diffusion.py)

A.6 Data Loading (data.py)

```python
"""
CATH dataset for FrameDiff: loads protein backbones, converts to SE(3) frames.

All coordinates are stored and returned in **nanometers**.
"""

import json
import math
import torch
import numpy as np
from torch.utils.data import Dataset, Sampler
from typing import Dict, List, Optional, Tuple


# ---------------------------------------------------------------------------
# Geometry helpers
# ---------------------------------------------------------------------------


def _gram_schmidt(v1: torch.Tensor, v2: torch.Tensor) -> torch.Tensor:
    """Gram-Schmidt orthogonalisation to build a rotation matrix from two vectors.

    Parameters
    ----------
    v1 : (..., 3)   first vector  (C - CA direction)
    v2 : (..., 3)   second vector (N - CA direction)

    Returns
    -------
    R : (..., 3, 3) rotation matrix with columns [e1, e2, e3]
    """
    e1 = v1 / (torch.linalg.norm(v1, dim=-1, keepdim=True) + 1e-8)
    # Remove e1 component from v2
    e2 = v2 - (v2 * e1).sum(-1, keepdim=True) * e1
    e2 = e2 / (torch.linalg.norm(e2, dim=-1, keepdim=True) + 1e-8)
    e3 = torch.cross(e1, e2, dim=-1)
    # R has e1,e2,e3 as columns: R = [e1 | e2 | e3]
    R = torch.stack([e1, e2, e3], dim=-1)                  # (..., 3, 3)
    return R


def _dihedral(a: torch.Tensor, b: torch.Tensor,
              c: torch.Tensor, d: torch.Tensor) -> torch.Tensor:
    """Compute dihedral angle for four points a-b-c-d.

    Returns angle in radians, shape (...).
    """
    b1 = b - a
    b2 = c - b
    b3 = d - c
    b2_norm = b2 / (torch.linalg.norm(b2, dim=-1, keepdim=True) + 1e-8)
    n1 = torch.cross(b1, b2, dim=-1)
    n2 = torch.cross(b2, b3, dim=-1)
    m1 = torch.cross(n1, b2_norm, dim=-1)
    x = (n1 * n2).sum(-1)
```

```
    y = (m1 * n2).sum(-1)
    return torch.atan2(y, x)


def _random_so3(shape: Tuple[int, ...], device="cpu", dtype=torch.float32) -> torch.Tensor:
    """Sample a uniform random SO(3) rotation matrix.

    Uses the QR decomposition approach (Mezzadri 2007).
    Returns (*shape, 3, 3).
    """
    n = 1
    for s in shape:
        n *= s
    z = torch.randn(n, 3, 3, device=device, dtype=dtype)
    q, r = torch.linalg.qr(z)
    # Ensure proper rotation (det = +1)
    sign = torch.sign(torch.diagonal(r, dim1=-2, dim2=-1))          # (n, 3)
    q = q * sign[:, None, :]
    det = torch.det(q)
    q[det < 0] = -q[det < 0]
    return q.reshape(*shape, 3, 3)


# ---------------------------------------------------------------------------
# Dataset
# ---------------------------------------------------------------------------


class CATHFrameDataset(Dataset):
    """CATH protein backbone dataset returning SE(3) frames.

    Each sample is a dictionary:
        rotations   : (N, 3, 3)  per-residue rotation matrices
        translations: (N, 3)     Calpha positions in nm (centered)
        psi_angles  : (N,)       psi torsion angles in rad
        mask        : (N,)       1.0 for valid residues, 0.0 for padding
        length      : int        actual chain length (unpadded)
    """

    def __init__(
        self,
        jsonl_path: str,
        splits_path: str,
        split: str = "train",
        max_len: int = 256,
        augment: bool = True,
    ):
        super().__init__()
        self.max_len = max_len
        self.augment = augment

        # Load split names
        with open(splits_path) as f:
            splits = json.load(f)
        valid_names = set(splits[split])

        # Load chains
        self.entries: List[Dict] = []
        with open(jsonl_path) as f:
```

```
            for line in f:
                rec = json.loads(line)
                if rec["name"] not in valid_names:
                    continue
                L = len(rec["seq"])
                if L > max_len or L < 10:
                    continue
                # Check for enough non-NaN coords
                ca = np.array(rec["coords"]["CA"])
                valid = ~np.isnan(ca).any(axis=1)
                if valid.sum() < 10:
                    continue
                self.entries.append(rec)

        print(f"[CATHFrameDataset] split={split}  loaded {len(self.entries)} chains "
              f"(max_len={max_len})")

    def __len__(self) -> int:
        return len(self.entries)

    def __getitem__(self, idx: int) -> Dict[str, torch.Tensor]:
        rec = self.entries[idx]
        L = len(rec["seq"])

        # 1. Extract backbone atom coordinates  (L, 3)  in Angstroms
        coords_N  = torch.tensor(rec["coords"]["N"],  dtype=torch.float32)
        coords_CA = torch.tensor(rec["coords"]["CA"], dtype=torch.float32)
        coords_C  = torch.tensor(rec["coords"]["C"],  dtype=torch.float32)
        coords_O  = torch.tensor(rec["coords"]["O"],  dtype=torch.float32)

        # Build validity mask: all four atoms must be non-NaN
        valid = (
            ~torch.isnan(coords_N).any(-1)
            & ~torch.isnan(coords_CA).any(-1)
            & ~torch.isnan(coords_C).any(-1)
            & ~torch.isnan(coords_O).any(-1)
        )  # (L,)

        # Replace NaN with 0 for computation safety
        coords_N  = torch.nan_to_num(coords_N,  0.0)
        coords_CA = torch.nan_to_num(coords_CA, 0.0)
        coords_C  = torch.nan_to_num(coords_C,  0.0)
        coords_O  = torch.nan_to_num(coords_O,  0.0)

        # 2. Convert to nanometers
        coords_N  = coords_N  / 10.0
        coords_CA = coords_CA / 10.0
        coords_C  = coords_C  / 10.0
        coords_O  = coords_O  / 10.0

        # 3. Center Calpha to zero mean (only valid residues)
        valid_f = valid.float()
        center = (coords_CA * valid_f[:, None]).sum(0) / valid_f.sum().clamp(min=1)
        coords_N  = coords_N  - center
        coords_CA = coords_CA - center
        coords_C  = coords_C  - center
        coords_O  = coords_O  - center

        # 4. Gram-Schmidt frames from N, CA, C
        v1 = coords_C - coords_CA        # (L, 3)
        v2 = coords_N - coords_CA        # (L, 3)
```

```
        rotations = _gram_schmidt(v1, v2)   # (L, 3, 3)

        # 5. Psi torsion angles: dihedral(N_i, CA_i, C_i, N_{i+1})
        psi = torch.zeros(L)
        if L > 1:
            psi[:-1] = _dihedral(coords_N[:-1], coords_CA[:-1],
                                 coords_C[:-1], coords_N[1:])
        # Last residue: no N_{i+1}, leave as 0

        # 6. Data augmentation: random SO(3) rotation of ALL atoms
        if self.augment:
            R_aug = _random_so3((), device=coords_CA.device, dtype=coords_CA
                .dtype)  # (3,3)
            coords_CA = torch.einsum("ij, nj -> ni", R_aug, coords_CA)
            coords_N  = torch.einsum("ij, nj -> ni", R_aug, coords_N)
            coords_C  = torch.einsum("ij, nj -> ni", R_aug, coords_C)
            coords_O  = torch.einsum("ij, nj -> ni", R_aug, coords_O)
            # Rotate frames: R_new = R_aug @ R_old
            rotations = torch.einsum("ij, njk -> nik", R_aug, rotations)

        translations = coords_CA    # (L, 3)

        # 7. Pad to max_len
        pad = self.max_len - L
        mask = torch.cat([valid_f, torch.zeros(pad)])

        if pad > 0:
            rotations    = torch.cat([rotations,    torch.eye(3).unsqueeze
                (0).expand(pad, 3, 3)], dim=0)
            translations = torch.cat([translations, torch.zeros(pad, 3)],
                dim=0)
            psi          = torch.cat([psi,          torch.zeros(pad)], dim
                =0)

        return {
            "rotations":    rotations,       # (max_len, 3, 3)
            "translations": translations,    # (max_len, 3)
            "psi_angles":   psi,             # (max_len,)
            "mask":         mask,             # (max_len,)
            "length":       L,
        }


#
    ----------------------------------------------------------------------------

# Length-bucketed sampler
#
    ----------------------------------------------------------------------------


class LengthBucketSampler(Sampler):
    """Groups proteins by length for efficient batching.

    Buckets: [10-50), [50-100), [100-150), [150-200), [200-256].
    Within each bucket, indices are shuffled; buckets are iterated in
    random order to mix the training signal.
    """

    def __init__(
        self,
        dataset: CATHFrameDataset,
        batch_size: int = 8,
```

```
        bucket_boundaries: Optional[List[int]] = None,
        shuffle: bool = True,
        drop_last: bool = False,
    ):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.drop_last = drop_last

        if bucket_boundaries is None:
            bucket_boundaries = [50, 100, 150, 200]
        self.boundaries = bucket_boundaries

        # Assign each sample to a bucket
        self.buckets: List[List[int]] = [[] for _ in range(len(self.
            boundaries) + 1)]
        for i, entry in enumerate(dataset.entries):
            L = len(entry["seq"])
            placed = False
            for bi, b in enumerate(self.boundaries):
                if L < b:
                    self.buckets[bi].append(i)
                    placed = True
                    break
            if not placed:
                self.buckets[-1].append(i)

    def __iter__(self):
        batches = []
        for bucket in self.buckets:
            if len(bucket) == 0:
                continue
            indices = list(bucket)
            if self.shuffle:
                perm = torch.randperm(len(indices)).tolist()
                indices = [indices[p] for p in perm]
            for i in range(0, len(indices), self.batch_size):
                batch = indices[i : i + self.batch_size]
                if self.drop_last and len(batch) < self.batch_size:
                    continue
                batches.append(batch)

        # Shuffle batch order
        if self.shuffle:
            perm = torch.randperm(len(batches)).tolist()
            batches = [batches[p] for p in perm]

        for batch in batches:
            yield batch

    def __len__(self) -> int:
        total = 0
        for bucket in self.buckets:
            n = len(bucket)
            if self.drop_last:
                total += n // self.batch_size
            else:
                total += (n + self.batch_size - 1) // self.batch_size
        return total
```

Listing 6: Data Loading (data.py)