Diffusion Models for Graph Signal Generation and Augmentation

Technical Report

January 8, 2026

Abstract

This report describes the application of diffusion models to generate realistic graph signals on fixed meshes. We develop two approaches: (1) a GNN-based denoiser with edge features for direction-aware message passing, and (2) a Graph-Aware Diffusion (GAD) approach using heat equation dynamics on the graph Laplacian. A key insight is that the diffusion framework naturally supports data augmentation by controlling the noise injection level, allowing generation of signals that are "similar but not identical" to training examples.

1 Introduction and Motivation

1.1 The Problem

Given a set of graph signals defined on a fixed mesh (e.g., vertex positions from physics simulations), we want to generate new signals that:

1. Look realistic (could plausibly come from the same physical process)
2. Are diverse (not identical to training examples)
3. Respect the underlying structure (mesh connectivity, boundary conditions)

1.2 Use Case: Data Augmentation

The primary application is **data augmentation** for training graph neural networks. When real simulation data is expensive to generate, we want to:

- Take a small set of real examples
- Generate many similar-but-different variations
- Use these to augment training data

1.3 Training Data

We use DeepMind's flag simulation dataset (MeshGraphNets):

- 50 simulations $\times$ 401 timesteps = **20,050 training frames**
- Each frame: 1,579 vertices $\times$ 3 coordinates (xyz positions)
- Fixed mesh connectivity: 3,028 triangles

Each frame is treated as an independent graph signal $\mathbf{x} \in \mathbb{R}^{V\times 3}$.

1.3.1 Custom Simulation (Optional)

We also provide a GPU-accelerated cloth simulation (`simulate_flag.py`) for generating additional training data:

- Position-based dynamics with vectorized constraint solving
- Chaotic wind forces with multi-frequency sinusoidal gusts
- Configurable temporal stride (default: 100 steps) ensures statistically independent frames
- Avoids the problem of highly correlated consecutive frames

```
python simulate_flag.py --record --stride 100 --frames 100000
```

2 Methods We Explored (and Why They Failed)

2.1 Approach 1: Spectral Noise Filtering

Filter white noise with the spectral envelope of real signals.

Result: Failed to capture boundary conditions (fixed pole moved freely), spatial coherence, and produced jittery outputs.

2.2 Approach 2: Spectral VAE

Variational autoencoder operating in the graph Fourier domain.

Result: On FEA data, outputs were nearly identical to inputs. Training data lacked diversity—generative models need varied examples to learn meaningful variation.

3 Diffusion Models: The Solution

3.1 Why Diffusion?

Diffusion models (Stable Diffusion, DALL-E 3) are state-of-the-art for generation because they:

- Learn implicit constraints from data
- Generate high-quality, diverse outputs
- Support **controllable augmentation** via partial denoising

3.2 The Diffusion Process

3.2.1 Forward Process (Adding Noise)

Given a clean signal $\mathbf{x}_0$, progressively add Gaussian noise:

$$\mathbf{x}_n = \sqrt{\bar{\alpha}_n}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_n}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, I) \tag{1}$$

where $\bar{\alpha}_n = \prod_{i=1}^{n} \alpha_i$ is the cumulative noise schedule, and n is the **noise step**.

At $n = 0$: signal is clean. At $n = N$: signal is pure noise.

3.2.2 Reverse Process (Denoising)

A neural network $\boldsymbol{\epsilon}_\theta$ learns to predict the noise:

$$\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_n, n) \tag{2}$$

3.2.3 Training

Algorithm 1 Diffusion Training

```
1: repeat
2:     Sample $\mathbf{x}_0 \sim$ training data (single frame)
3:     Sample $n \sim \text{Uniform}(1, N)$
4:     Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$
5:     $\mathbf{x}_n \leftarrow \sqrt{\bar{\alpha}_n}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_n}\,\boldsymbol{\epsilon}$
6:     Gradient step on $\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_n, n)\|^2$
7: until converged
```

4 Key Insight: Augmentation via Partial Denoising

Instead of starting from pure noise ($n = N$), start from a **real signal with added noise**:

Algorithm 2 Signal Augmentation via Partial Denoising

```
1: Input: Real signal $\mathbf{x}_0$, starting noise step $n^*$
2: Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$
3: $\mathbf{x}_{n^*} \leftarrow \sqrt{\bar{\alpha}_{n^*}}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{n^*}}\,\boldsymbol{\epsilon}$
4: for $n = n^*, n^*-1, \ldots, 1, 0$ do
5:     $\hat{\boldsymbol{\epsilon}} \leftarrow \boldsymbol{\epsilon}_\theta(\mathbf{x}_n, n)$
6:     $\mathbf{x}_{n-1} \leftarrow \text{DenoisingStep}(\mathbf{x}_n, \hat{\boldsymbol{\epsilon}}, n)$
7: end for
8: return $\mathbf{x}_0$                                   ▷ Augmented signal (similar to input)
```

4.1 Controlling Similarity

The starting noise step n^* controls how different the output is from the input:

n^*	Noise Level	Result
1000	Maximum	Completely new signal
700	High	Loosely similar
500	Medium	Same structure, different details
300	Low	Very close to original
0	None	Identical to input

5 Two Denoiser Architectures

We implement two approaches for comparison.

5.1 Approach A: GNN with Edge Features (Ours)

5.1.1 Edge Features for Direction Awareness

Standard GNN message passing treats all neighbors identically:

$$h_i^{(l+1)} = h_i^{(l)} + \text{Aggregate}_{j \in \mathcal{N}(i)} \text{MLP}(h_i^{(l)}, h_j^{(l)}) \tag{3}$$

This is **isotropic**—unlike CNN kernels which have different weights for different positions.

We add **edge features** encoding the relative position of each neighbor:

$$\mathbf{e}_{ij} = [\mathbf{p}_j - \mathbf{p}_i, \|\mathbf{p}_j - \mathbf{p}_i\|] \in \mathbb{R}^4 \tag{4}$$

where $\mathbf{p}_i, \mathbf{p}_j$ are vertex positions in the rest state.

The message function becomes:

$$m_{j \to i} = \text{MLP}(h_i, h_j, \mathbf{e}_{ij}) \tag{5}$$

This allows the network to learn **direction-dependent** filters, similar to CNN kernels.

5.1.2 Architecture

Component	Description
Input projection	Linear: $3 \to D$
Encoder	4 MeshConv layers with residual connections
Timestep embedding	Sinusoidal + MLP
Decoder	4 MeshConv layers with residual connections
Output projection	Linear: $D \to 3$

Parameters: ~534K. Hidden dimension $D = 128$.

5.2 Approach B: Graph-Aware Diffusion (GAD)

Based on Rozada et al. (arXiv:2510.05036).

5.2.1 Graph-Aware Forward Process

Instead of isotropic Gaussian noise, GAD uses the **heat equation** on the graph:

$$d\mathbf{x}_t = -c_t \mathbf{L}_\gamma \mathbf{x}_t \, dt + \sqrt{2c_t}\sigma \, d\mathbf{w}_t \tag{6}$$

where $\mathbf{L}$ is the graph Laplacian and $\mathbf{L}_\gamma = \mathbf{L} + \gamma \mathbf{I}$.

Key property: Noise spreads along graph edges. Smooth signals (low graph frequency) decay slower than high-frequency signals.

5.2.2 Polynomial Graph Filter Denoiser

The denoiser is a polynomial in the Laplacian:

$$H(\mathbf{L}) = \sum_{k=0}^{K} \theta_k \mathbf{L}^k \tag{7}$$

Each power $\mathbf{L}^k$ captures k-hop neighborhoods. The learnable coefficients θ_k weight contributions from different hop distances.

5.3 Comparison

Aspect	GNN + Edge Features	GAD
Forward process	Isotropic Gaussian	Heat equation on $\mathbf{L}$
Denoiser	MLP message passing	Polynomial filter $H(\mathbf{L})$
Edge features	Yes (direction vectors)	No (uses $\mathbf{L}^k$)
Parameters	$\sim$534K	$\sim$100K
Interpretability	Black-box MLP	Spectral (graph frequencies)

5.4 Spectral Analysis: High-Pass vs Low-Pass

A key difference between the two approaches lies in how they handle graph frequencies.

5.4.1 The Graph Laplacian as a High-Pass Filter

The Laplacian measures how different each vertex is from its neighbors:

$$(\mathbf{L}\mathbf{x})_i = \sum_{j \in \mathcal{N}(i)} (x_i - x_j) \tag{8}$$

The eigenvalues λ of $\mathbf{L}$ correspond to graph frequencies:

- $\lambda = 0$: constant signal (DC component)
- λ small: smooth, slowly-varying signals (low frequency)
- λ large: rapidly-varying signals (high frequency)

5.4.2 GAD's Forward Process Destroys High Frequencies First

The heat kernel solution is:

$$\mathbf{x}(t) = \exp(-t\mathbf{L})\mathbf{x}(0) \tag{9}$$

In the spectral domain, each eigenvalue λ is attenuated by $\exp(-t\lambda)$:

- Small λ (low freq): $\exp(-t\lambda) \approx 1$ — **preserved**
- Large λ (high freq): $\exp(-t\lambda) \approx 0$ — **destroyed**

This is inherently a **low-pass** forward process. By the time t is large (high noise), the high-frequency components are completely gone. The denoiser cannot recover information that has been erased.

5.4.3 Our Approach Preserves All Frequencies

With isotropic Gaussian noise:

$$\mathbf{x}_n = \sqrt{\bar{\alpha}_n}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_n}\,\boldsymbol{\epsilon} \tag{10}$$

All frequencies are corrupted equally—both low and high frequency components are scaled by $\sqrt{\bar{\alpha}_n}$ and buried under noise. But they are still **present** in the noisy signal, not destroyed. The denoiser can learn to recover both.

5.4.4 Implications

	GAD (Heat Equation)	Ours (Isotropic Noise)
Forward process	Low-pass	All-pass
High freq at $n = N$	Gone (irrecoverable)	Buried but present
Denoiser task	Recover low freq, hallucinate high freq	Recover all freq
Best for	Smooth signals	Signals with fine detail

For mesh data with sharp features (edges, creases, boundaries), our isotropic approach may be preferable since it doesn't preferentially destroy high-frequency geometric detail.

6 Implementation

6.1 Data Pipeline

1. Load flag simulation data (50 trajectories $\times$ 401 frames)
2. Flatten to 20,050 individual frames
3. Normalize to $[-1, 1]$ range via min-max scaling (standard for DDPM)
4. Split 90% train, 10% validation

6.2 Training Configuration

Parameter	Value
Batch size	32
Learning rate	10^{-4}
Optimizer	AdamW
Noise steps	1000
Schedule	Cosine
Epochs	100
Mixed precision	AMP (on CUDA)
Gradient clipping	1.0

Mixed precision training via PyTorch AMP (Automatic Mixed Precision) is enabled on CUDA devices for faster training with reduced memory usage.

6.3 Usage

```
python setup_flag_data.py                          # Download and prepare data
python simulate_flag.py --record --stride 100      # (Optional) Generate more data
python train_flag_diffusion.py                     # Train model
```

7 Relationship to Prior Work

This work combines several established methods. We explicitly document the provenance of each component to clarify that no novel algorithms are introduced—our contribution is the application of these methods to graph signal augmentation.

7.1 Component Attribution

Our Component	Established Method	Reference
Forward/reverse diffusion process	DDPM	Ho et al. [1]
Cosine noise schedule	Improved DDPM	Nichol & Dhariwal [2]
Partial denoising for augmentation	SDEdit	Meng et al. [7]
Message passing with edge features	MeshGraphNets / MPNN	Pfaff et al. [5], Gilmer et al. [3]
Sinusoidal timestep embedding	Transformer / DDPM	Vaswani et al., Ho et al. [1]
Graph Laplacian analysis	Graph Signal Processing	Shuman et al. [6]
Heat equation forward process	GAD	Rozada et al. [4]
Polynomial graph filters	GAD / Spectral GNNs	Rozada et al. [4]

7.2 How We Use Each Method

7.2.1 DDPM (Ho et al., 2020)

We use the standard DDPM formulation unchanged:

- Forward process: $\mathbf{x}_n = \sqrt{\bar{\alpha}_n}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_n}\boldsymbol{\epsilon}$
- Training objective: $\mathcal{L} = \mathbb{E}[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_n, n)\|^2]$
- Reverse sampling: iterative denoising from $n = N$ to $n = 0$

The only difference is that our signal $\mathbf{x} \in \mathbb{R}^{V\times 3}$ is a graph signal rather than an image.

7.2.2 Improved DDPM / Cosine Schedule (Nichol & Dhariwal, 2021)

We use the cosine schedule exactly as proposed:

$$\bar{\alpha}_n = \frac{f(n)}{f(0)}, \quad f(t) = \cos\left(\frac{t/N + s}{1+s}\cdot\frac{\pi}{2}\right)^2 \tag{11}$$

with $s = 0.008$. This provides smoother noise progression than linear schedules.

7.2.3 SDEdit (Meng et al., 2022)

Our augmentation procedure is SDEdit applied to graph signals:

1. Start with real signal $\mathbf{x}_0$
2. Add noise to intermediate step n^*: $\mathbf{x}_{n^*} = \sqrt{\bar{\alpha}_{n^*}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{n^*}}\boldsymbol{\epsilon}$
3. Denoise from n^* back to 0

This is identical to SDEdit's “img2img” procedure, substituting graph signals for images.

7.2.4 MeshGraphNets / MPNN (Pfaff et al., 2021; Gilmer et al., 2017)

Our GNN denoiser uses the standard message passing framework:

$$m_{j\to i} = \phi_e(h_i, h_j, \mathbf{e}_{ij}), \quad h_i' = \phi_v\left(h_i, \sum_{j\in\mathcal{N}(i)} m_{j\to i}\right) \tag{12}$$

where ϕ_e and ϕ_v are MLPs. Edge features $\mathbf{e}_{ij} = [\mathbf{p}_j - \mathbf{p}_i, \|\mathbf{p}_j - \mathbf{p}_i\|]$ encode relative positions, exactly as in MeshGraphNets.

7.2.5 GAD (Rozada et al., 2025)

For comparison, we implement GAD's approach:

- Heat equation forward process on the graph Laplacian
- Polynomial graph filter denoiser: $H(\mathbf{L}) = \sum_k \theta_k \mathbf{L}^k$

7.3 What Is Not Novel

To be explicit, we do **not** claim novelty for:

- The diffusion framework (DDPM)
- The noise schedule (Improved DDPM)
- The partial denoising augmentation strategy (SDEdit)
- The message passing architecture (MPNN/MeshGraphNets)
- The graph signal processing perspective (Shuman et al.)

7.4 Our Contribution

Our contribution is **applying** these established methods to:

1. Generate realistic graph signals on fixed meshes
2. Provide controllable data augmentation for training physics-based GNNs
3. Compare isotropic (DDPM) vs graph-aware (GAD) forward processes

This is an **application paper**, demonstrating that standard diffusion techniques transfer effectively to the graph signal domain.

8 Summary

1. **Simple approaches failed**: Spectral filtering and VAEs couldn't capture the full structure of graph signals.
2. **Diffusion learns holistically**: By learning to denoise at all noise levels, the model captures spatial coherence and physical constraints implicitly.

3. **Augmentation is built-in**: Partial denoising provides controllable augmentation—start from intermediate noise level to generate similar-but-different signals.

4. **Two architectures**:

   - GNN with edge features: direction-aware, like CNN kernels
   - GAD: graph-aware noise via heat equation, polynomial filter denoiser

References

[1] Ho, J., Jain, A., & Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. *NeurIPS*.

[2] Nichol, A. & Dhariwal, P. (2021). Improved Denoising Diffusion Probabilistic Models. *ICML*.

[3] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. *ICML*.

[4] Rozada, S., et al. (2025). Graph-Aware Diffusion for Signal Generation. *arXiv:2510.05036*.

[5] Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., & Battaglia, P. W. (2021). Learning Mesh-Based Simulation with Graph Networks. *ICLR*.

[6] Shuman, D. I., et al. (2013). The Emerging Field of Signal Processing on Graphs. *IEEE Signal Processing Magazine*.

[7] Meng, C., et al. (2022). SDEdit: Guided Image Synthesis and Editing with Stochastic Differential Equations. *ICLR*.