Deep Learning Volatility

A Modular Framework for Learning and Calibrating Implied Volatility Surfaces

Bianchi Giacomo

August 28, 2025

Contents

Chapter 1

Introduction

This document describes the *Deep Learning Volatility* (DLV) framework: a modular system to learn, generate, and calibrate implied volatility (IV) surfaces for classical and rough stochastic volatility models. DLV follows a two-step design: (i) an *offline* learning phase where neural networks approximate expensive pricing maps, and (ii) an *online* calibration phase where the surrogate is optimized against market data. Our approach bridges grid-based surface learning (Horvath, Muguruza, and Tomas, 2021) with pointwise training on *random grids/smiles* (Baschetti, Bormetti, and Rossi, 2024). We target modern rough models (e.g., rough Heston, rough Bergomi), where direct Monte Carlo or fractional-Riccati methods hinder calibration speed.

Contributions. This documentation

- formalizes the architecture (Chapter 2),
- specifies the stochastic-process interface used by all models (Chapter 3),
- details the dataset generation via Monte Carlo on *random grids/smiles* (Chapter 4),
- describes the neural pricers (grid, pointwise, multi-regime) and interpolation/repair modules (Chapter 5), and
- outlines calibration workflows and reproducibility (Chapter 6).

Intended audience. Quantitative finance students, quant researchers and engineers extending DLV to new models, building datasets, or integrating calibration into production.

Chapter 2

System Overview

DLV is organized into three cooperating subsystems:

S1. **Process layer** — a unified `StochasticProcess` protocol with a factory for instantiation and shared utilities.

S2. **Data layer** — a `DatasetBuilder` that samples parameters, simulates paths, and constructs training pairs on *random grids/smiles*.

S3. **Learning layer** — neural pricers mapping model parameters to IV values (whole grid or single points), plus tools for smooth interpolation and smile repair.

The core abstractions live in code modules such as `pricer.py` and `stochastic_interface.py`. See Chapter 3 and Chapter 5.

2.1 Framework Dependencies and Integration

2.1.1 PFHedge Integration and Process Implementations

DLV builds upon the robust stochastic process implementations from PFHedge[1], a comprehensive framework for derivative hedging developed by *"Preferred Networks"*. The majority of classical stochastic process generators (Heston, geometric Brownian motion, jump diffusion models, etc.) are adapted from PFHedge's well-tested implementations, ensuring numerical stability and computational efficiency.

Custom Rough Heston Implementation While most classical processes leverage PFHedge implementations, the rough Heston model represents a significant extension developed specifically for DLV. This implementation follows the Hybrid Quadratic Exponential (HQE) scheme detailed in Bertolo (2024):

```
def generate_rough_heston(n_paths: int, n_steps: int, H: float = 0.1,
       nu: float = 0.3, rho: float = -0.7,
       kappa: float = 0.3, theta: float = 0.02, ...):
       """
       Rough Heston implementation using HQE scheme.
       Reference: Bertolo (2024), Section 3.2.
       """
       # Forward variance curve computation
       xi = compute_forward_variance_curve(V0, theta, kappa, H, nu, n_steps, dt)

       # Vectorized QE scheme for non-negative variance generation
       u_n, chi_n = vectorized_qe_bivariate(mean_shifted, mean_shifted,
       var_u, var_chi, cov_u_chi, Z1, Z2)
       ...
```

[1] `https://github.com/pfnet-research/pfhedge`

This implementation incorporates several key innovations:

- **Vectorized kernel computations**: Efficient calculation of power-law kernel integrals required for rough volatility dynamics
- **Antithetic variable support**: Variance reduction techniques fully integrated with the HQE scheme
- **Numerical stability**: Robust handling of extreme roughness parameters through adaptive step sizing and clamping

2.1.2 Bidirectional Framework Compatibility

The modular design of DLV enables seamless integration with PFHedge for users interested in derivative hedging applications. The unified `StochasticProcess` interface ensures that neural pricers trained within DLV can be directly utilized within PFHedge's hedging framework:

```
# Example: Using DLV-trained neural pricer for PFHedge hedging
from pfhedge import BlackScholes, EuropeanOption
from deepLearningVolatility.nn.pricer import PointwiseNetworkPricer

# Load trained DLV neural pricer
neural_pricer = PointwiseNetworkPricer.load('trained_model.pt')

# Integrate with PFHedge hedging simulation
def neural_pricing_function(spot, vol, time_to_maturity, strike):
       theta = extract_model_parameters(vol) # Convert vol to model params
       T = torch.tensor([time_to_maturity])
       k = torch.log(torch.tensor([strike/spot]))
       return neural_pricer.price_iv(theta, T, k)

# Use in PFHedge derivative hedging
derivative = EuropeanOption(strike=100, maturity=0.25)
hedger = SomeHedgingStrategy(derivative, pricing_fn=neural_pricing_function)
```

2.1.3 Extension Opportunities

This compatibility opens several research and practical directions:

- **Neural delta hedging**: Using neural-approximated Greeks for real-time hedging decisions
- **Rough model hedging**: Applying sophisticated rough volatility models in practical hedging scenarios
- **Calibration-hedging loops**: Real-time model recalibration integrated with hedging workflows

The foundation provided by PFHedge ensures that DLV maintains compatibility with established quantitative finance workflows while extending capabilities into neural acceleration and rough volatility modeling. Users can thus benefit from both frameworks' strengths: PFHedge's proven hedging methodologies and DLV's advanced neural approximation techniques.

Chapter 3

Stochastic-Process Interface

DLV exposes stochastic models through a single protocol and registers them behind a factory so you can construct processes by key and simulate paths in a uniform way. This chapter covers: (i) what is implemented, (ii) the common interface shared by all processes, and (iii) how simulations are invoked and what they return.

3.1 Architecture Overview

The stochastic process layer consists of three key components:

Core Abstractions

- `StochasticProcess` Protocol — Defines the contract all processes must implement
- `BaseStochasticProcess` — Abstract base class providing shared functionality
- `SimulationOutput` — Type-safe container for simulation results

Wrapper Pattern Implementation Each stochastic model is implemented as a thin wrapper that:

- Implements the unified `StochasticProcess` interface
- Forwards to vectorized generators (e.g., `generate_heston`, `generate_rough_bergomi`)
- Provides model-specific metadata and validation
- Handles device/dtype propagation and optional features

Factory Registry The `ProcessFactory` enables dynamic process creation by string keys with support for aliases, enabling flexible configuration and extensibility.

3.2 What's Implemented (Factory-Registered)

All models are exposed through thin *wrappers* that derive from `BaseStochasticProcess`; wrappers forward to the vectorized generators and implement the unified API (parameters, validation, simulation, absorption handling).

Model	Factory Key	θ (parameters)	Key Features
Brownian	`"brownian"`	(μ, σ)	Drift + constant vol
Geometric Brownian	`"geometric_brownian"`	$(\mu,\ \sigma)$	No absorption, constant vol
Heston	`"heston"`	$(\kappa,\ \theta,\ \sigma,\ \rho)$	Stochastic vol, CIR variance
Rough Heston	`"rough_heston"`	$(H,\ \nu,\ \rho,\ \kappa,\ \theta_{\text{var}})$	Fractional vol, memory
Rough Bergomi	`"rough_bergomi"`	$(H,\ \eta,\ \rho,\ \xi_0)$	Forward variance curve
CIR	`"cir"`	$(\kappa,\ \theta,\ \sigma)$	Square-root diffusion
Vasicek (OU)	`"vasicek"`	$(\kappa,\ \theta,\ \sigma)$	Mean reversion
Merton jump	`"merton_jump"`	$(\mu,\ \sigma,\ \lambda,\ \mu_J,\ \sigma_J)$	Compound Poisson jumps
Kou double-exp	`"kou_jump"`	$(\mu,\ \sigma,\ \lambda,\ p,\ \eta_\uparrow,\ \eta_\downarrow)$	Asymmetric jump sizes

Aliases and registration. Most processes support multiple aliases (e.g., `"gbm"`, `"black_scholes"` for Geometric Brownian). The factory resolves all aliases to the same underlying class while maintaining a canonical key for configuration serialization.

3.3 Common Interface

All processes implement the same protocol and usually inherit shared behavior from `BaseStochasticProcess`. The interface consists of the following components:

3.3.1 Parameter Metadata

Each process exposes `num_params` and a `param_info` structure with *names*, *bounds*, and *defaults*:

```
@property
def param_info(self) -> ParameterInfo:
return ParameterInfo(
names=['kappa', 'theta', 'sigma', 'rho'],
bounds=[(0.1, 5.0), (0.01, 0.5), (0.1, 1.0), (-0.95, 0.95)],
defaults=[1.0, 0.04, 0.2, -0.7],
descriptions=['Mean reversion speed', 'Long-term variance',
'Volatility of variance', 'Correlation']
)
```

Validation is centralized via `validate_theta(theta)` which checks parameter count and bounds compliance.

3.3.2 Capability Flags

- **Absorption support.** `supports_absorption` indicates if the process can reach zero (important for barrier options and path-dependent payoffs).
- **Variance state.** `requires_variance_state` indicates if initialization needs both spot and variance values (true for stochastic volatility models).

3.3.3 Unified Simulation Entry Point

All processes expose a single `simulate` method:

$$\texttt{simulate}(\theta,\ n\_paths,\ n\_steps,\ dt,\ \texttt{init\_state} = \ldots,\ \texttt{device} = \ldots,\ \texttt{dtype} = \ldots,\ \texttt{antithetic} = False,\ **\texttt{kwargs})$$

This method allocates tensors on the requested device/dtype, validates θ, prepares an initial state, and forwards arguments to the model's generator. Device/dtype propagation ensures simulations run consistently on CPU/GPU. When supported by the underlying generator, `antithetic=True` reduces Monte Carlo variance at low cost.

3.4 Simulation I/O Contract

Every `simulate(...)` returns a typed container with consistent shapes:

```
class SimulationOutput(NamedTuple):
spot: Tensor # (n_paths, n_steps)
variance: Optional[Tensor] # (n_paths, n_steps) if available
auxiliary: Optional[Dict[str, Tensor]] # diagnostics/metadata
```

Shape guarantees. Spot and variance are always (N, T) when present, where $N =$ `n_paths` and $T =$ `n_steps`.

Auxiliary data. Wrappers may attach helpful extras such as instantaneous volatility, jump counts, or echoed hyperparameters. For example, Heston returns `auxiliary['volatility'] = sqrt(variance)`, while Rough Heston includes the roughness parameter `auxiliary['H']`.

Absorption awareness. Downstream pricers can reuse `handle_absorption` for consistent masking of absorbed paths across all models that support it.

3.5 Wrapper Implementation Pattern

The following example demonstrates the typical wrapper implementation using the Heston process:

```
class HestonProcess(BaseStochasticProcess):
def __init__(self, spot: float = 1.0):
super().__init__(spot)

@property
def num_params(self) -> int:
return 4

@property
def param_info(self) -> ParameterInfo:
return ParameterInfo(
names=['kappa', 'theta', 'sigma', 'rho'],
bounds=[(0.1, 5.0), (0.01, 0.5), (0.1, 1.0), (-0.95, 0.95)],
defaults=[1.0, 0.04, 0.2, -0.7],
descriptions=['Mean reversion speed', 'Long-term variance',
'Volatility of variance', 'Correlation']
)

@property
def supports_absorption(self) -> bool:
return True # Heston variance can reach zero

@property
def requires_variance_state(self) -> bool:
return True

def get_default_init_state(self) -> Tuple[float, ...]:
"""Use long-term variance as default initial variance."""
return (self.spot, self.param_info.defaults[1]) # (S0, V0)

def simulate(self, theta, n_paths, n_steps, dt, **kwargs):
# 1. Parameter validation
is_valid, error_msg = self.validate_theta(theta)
if not is_valid:
raise ValueError(f"Invalid parameters: {error_msg}")
```

```
        # 2. Extract parameters and forward to generator
        kappa, theta_param, sigma, rho = theta.tolist()
        result = generate_heston(
        n_paths=n_paths, n_steps=n_steps,
        kappa=kappa, theta=theta_param, sigma=sigma, rho=rho,
        dt=dt, device=kwargs.get('device'), dtype=kwargs.get('dtype')
        )

        # 3. Return standardized output
        return SimulationOutput(
        spot=result.spot,
        variance=result.variance,
        auxiliary={'volatility': torch.sqrt(result.variance)}
        )
```

3.5.1 Key Design Principles

1. **Separation of concerns** — Wrappers handle interface compliance; generators handle numerical computation.
2. **Parameter validation** — Centralized bounds checking with descriptive error messages.
3. **Device/dtype propagation** — Consistent GPU/CPU and precision handling across all models.
4. **Default state management** — Sensible defaults for initial conditions (e.g., long-run variance for stochastic volatility models).
5. **Error handling** — Comprehensive validation with actionable error messages.

3.6 Absorption Handling

Processes that can reach zero implement specialized absorption logic through the `handle_absorption` method:

```
        def handle_absorption(self, paths: Tensor, dt: float,
        threshold: float = 1e-10) -> Tuple[Tensor, Tensor]:
        """
        Returns:
        absorption_times: When each path first hits zero
        absorbed_mask: Boolean mask of absorbed paths
        """
```

The default implementation efficiently identifies first-hitting times using cumulative sums and tensor operations, avoiding explicit loops:

```
        zero_mask = paths <= threshold
        cumsum = zero_mask.cumsum(dim=1)
        first_zero_mask = (cumsum == 1) & zero_mask

        padded_mask = torch.cat([first_zero_mask,
        torch.ones(paths.shape[0], 1, dtype=torch.bool,
        device=paths.device)], dim=1)

        absorption_indices = padded_mask.to(torch.float32).argmax(dim=1)
        absorption_times = absorption_indices.float() * dt
        absorbed_mask = absorption_indices < paths.shape[1]
```

Models like Rough Heston may override this method with model-specific thresholds adapted to their roughness parameter.

3.7 Factory Pattern and Process Creation

3.7.1 Process Registration

New processes are registered with the factory along with optional aliases:

```
# Register with primary key
ProcessFactory.register('heston', HestonProcess)

# Register with aliases
ProcessFactory.register('rough_heston', RoughHestonProcess,
aliases=['roughheston', 'rough-heston'])
```

3.7.2 Dynamic Process Creation

Processes are instantiated by string key, enabling configuration-driven model selection:

```
# Create by canonical key (case-insensitive)
process = ProcessFactory.create("heston", spot=100.0)

# Works with aliases
process = ProcessFactory.create("roughheston", spot=100.0)

# List all available processes and aliases
available = ProcessFactory.list_available()
```

3.7.3 Example Usage

The following code demonstrates typical usage patterns:

```
from deepLearningVolatility.stochastic.stochastic_interface import ProcessFactory
import torch

# 1) Create a process by key
proc = ProcessFactory.create("heston", spot=1.0)
theta = torch.tensor([0.5, 0.04, 0.3, -0.7])

# 2) Simulate paths
out = proc.simulate(
theta=theta, n_paths=8192, n_steps=256, dt=1/365,
device=torch.device("cuda"), dtype=torch.float32, antithetic=True
)

# 3) Use results
S = out.spot # (N, T) spot paths
V = out.variance # (N, T) variance paths (if available)
vol = out.auxiliary['volatility'] # instantaneous volatility
```

3.8 Extensibility

Adding a new stochastic process to DLV requires implementing the wrapper and registering it with the factory:

1. **Implement the wrapper**:

```
class MyNewProcess(BaseStochasticProcess):
@property
def param_info(self) -> ParameterInfo:
return ParameterInfo(names=[...], bounds=[...], defaults=[...])

def simulate(self, theta, n_paths, n_steps, dt, **kwargs):
# Forward to numerical generator
result = generate_my_new_process(...)
return SimulationOutput(spot=result.spot, ...)
```

2. **Create the numerical generator** (in separate module):

```
def generate_my_new_process(n_paths, n_steps, param1, param2, ...):
# Efficient vectorized implementation
paths = ... # Monte Carlo or analytical solution
return SimulationOutput(spot=paths, variance=None)
```

3. **Register with factory**:

```
ProcessFactory.register('my_new_process', MyNewProcess,
aliases=['mynew', 'my-new'])
```

This modular design ensures that new models integrate seamlessly with the existing neural pricing infrastructure, dataset generation pipelines, and calibration workflows without requiring changes to client code.

Chapter 4

Data Generation for Neural Volatility Pricing

Training neural networks for volatility surface approximation requires generating large-scale, high-quality datasets that capture the complex relationships between model parameters and implied volatilities. This chapter describes DLV's data generation infrastructure, which implements three distinct approaches from the quantitative finance literature: the grid-based method of Horvath, Muguruza, and Tomas (2021), the pointwise approach of Baschetti, Bormetti, and Rossi (2024), and multi-regime specializations designed to address the varying complexity of volatility surfaces across different maturity and moneyness regimes.

The core component, `DatasetBuilder`, orchestrates parameter sampling, Monte Carlo simulations, and data normalization while providing consistent interfaces across all stochastic processes. Advanced features include checkpoint management for long-running computations, memory-optimized batch processing, and adaptive Monte Carlo parameters tuned for each process type.

4.1 Overview of Neural Volatility Calibration Approaches

Modern neural network approaches to volatility model calibration can be categorized into two fundamental paradigms, each with distinct advantages and computational trade-offs.

4.1.1 Grid-Based Approach (Horvath, Muguruza, and Tomas (2021))

The pioneering work of Horvath, Muguruza, and Tomas (2021) introduced the grid-based approach, which treats implied volatility surfaces as "collections of pixels" over a two-dimensional grid in strike and time to maturity. The neural network learns a mapping from model parameters $\boldsymbol{\theta}$ to complete volatility grids:

$$F^M : \boldsymbol{\theta} \mapsto \{IV(T_i, K_j)\}_{i=1,\dots,n}^{j=1,\dots,m} \tag{4.1}$$

where $\{T_i\}$ and $\{K_j\}$ represent fixed maturity and strike grids. Training minimizes the loss:

$$\mathcal{L}_{\text{grid}} = \sum_{u=1}^{N} \sum_{i=1}^{n} \sum_{j=1}^{m} \left(F^M(\boldsymbol{\theta}_u; \mathbf{w})_{ij} - \sigma_{BS}^M(\boldsymbol{\theta}_u)_{ij} \right)^2 \tag{4.2}$$

This approach offers several advantages: (i) efficient training on relatively small networks (4 hidden layers, 30 nodes each), (ii) natural representation of volatility surfaces as image-like data, and (iii) fast calibration once trained. However, it requires two rounds of interpolation/extrapolation: first to project market data onto the training grid, then to evaluate the trained network at arbitrary market points.

4.1.2 Pointwise Approach with Random Grids (Baschetti, Bormetti, and Rossi (2024))

Baschetti, Bormetti, and Rossi (2024) addressed the interpolation limitations by developing a pointwise approach that learns individual volatility points rather than complete grids:

$$F^P : (\boldsymbol{\theta}, T, K) \mapsto IV(T, K) \tag{4.3}$$

The key innovation lies in training data generation using *random grids*: for each parameter vector $\boldsymbol{\theta}_u$, maturities are sampled from temporal buckets and strikes follow adaptive ranges:

$$T \sim \text{Uniform(bucket)} \quad \text{for buckets } \{[0.003, 0.030], [0.030, 0.090], \ldots\} \tag{4.4}$$

$$K \in [S_0(1 - l\sqrt{T}), S_0(1 + u\sqrt{T})] \quad \text{with } l = 0.55, u = 0.30 \tag{4.5}$$

This "random grid" generation is computationally efficient: pricing an entire smile at maturity T costs the same as pricing a single strike at that maturity, since the same Monte Carlo paths can be reused across strikes.

4.1.3 Multi-Regime Specialization

A critical insight from empirical volatility modeling is that different maturity regimes exhibit vastly different sensitivities to model parameters, particularly for rough volatility models. Short-term options ($\leq$1 month) are highly sensitive to roughness parameters and exhibit extreme moneyness effects, while long-term options ($\geq$1 year) are more sensitive to long-run variance levels and correlation parameters.

DLV addresses this through specialized multi-regime datasets that train separate networks for:

- **Short-term regime**: $T \in [\frac{1}{365}, \frac{30}{365}]$ years, focused on roughness and short-term skew
- **Mid-term regime**: $T \in (\frac{30}{365}, 1.0)$ years, capturing volatility-of-volatility effects
- **Long-term regime**: $T \in [1.0, 5.0]$ years, emphasizing mean reversion and correlation

4.2 DatasetBuilder Architecture

4.2.1 Core Components

The `DatasetBuilder` follows a modular design that adapts to any process implementing the `StochasticProcess` interface:

```
class DatasetBuilder:
def __init__(self, process, device='cpu', output_dir=None, dataset_type='train'):
       # Create process from string or use existing instance
       if isinstance(process, str):
              self.process = ProcessFactory.create(process)
       else:
              self.process = process

       # Extract process-specific parameter bounds and defaults
       self._update_param_bounds()

       # Initialize normalization statistics
       self.theta_mean = None
       self.theta_std = None
       self.iv_mean = None
       self.iv_std = None
```

4.2.2 Parameter Sampling Strategy

The builder uses Latin Hypercube Sampling (LHS) as the primary method for parameter space exploration. LHS provides superior space-filling properties compared to pure random sampling, ensuring better coverage of the parameter space with fewer samples:

```
def sample_theta_lhs(self, n_samples, seed=None):
    bounds = np.array([self.param_bounds[name] for name in self.param_names])
    sampler = qmc.LatinHypercube(d=self.process.num_params, seed=seed)
    unit_samples = sampler.random(n=n_samples)
    scaled_samples = qmc.scale(unit_samples, bounds[:, 0], bounds[:, 1])
    return torch.tensor(scaled_samples, dtype=torch.float32, device=self.device)
```

4.2.3 Process-Optimized Monte Carlo Parameters

Different stochastic processes require different Monte Carlo configurations for accurate and efficient pricing. The builder automatically selects optimized parameters:

```
def get_process_specific_mc_params(self, base_n_paths=30000):
    process_name = self.process.__class__.__name__.lower()

    if 'rough' in process_name:
        # Rough models need more paths for convergence
        return {
                    'n_paths': int(base_n_paths),
                    'use_antithetic': True,
                    'adaptive_dt': True,
                    'control_variate': True
            }
        elif 'heston' in process_name:
                return {
                    'n_paths': base_n_paths,
                    'use_antithetic': True,
                    'adaptive_dt': True,
                    'control_variate': True
                }
        # ... other process-specific configurations
```

4.3 Grid-Based Dataset Generation

4.3.1 Fixed Grid Implementation

Following Horvath, Muguruza, and Tomas (2021), the grid-based approach generates complete volatility surfaces on predetermined grids:

```
def build_grid_dataset(self, pricer: GridNetworkPricer, n_samples,
        n_paths=30000, normalize=True):
        # Sample parameters using LHS
        thetas = self.sample_theta_lhs(n_samples)

        # Get process-optimized MC parameters
        mc_params = self.get_process_specific_mc_params()

        # Generate IV surfaces for each parameter set
        ivs = []
        for theta in tqdm(thetas, desc="Building grid dataset"):
                iv = pricer._mc_iv_grid(
                    theta,
                    n_paths=mc_params['n_paths'],
                    use_antithetic=mc_params['use_antithetic'],
```

```
                        adaptive_dt=mc_params['adaptive_dt'],
                        control_variate=mc_params['control_variate']
                        )
                ivs.append(iv.cpu())

        iv_tensor = torch.stack(ivs).to(self.device)

        if normalize:
                self.compute_normalization_stats(thetas, iv_tensor)
                return self.normalize_theta(thetas), self.normalize_iv(iv_tensor)

        return thetas, iv_tensor
```

4.3.2 Checkpoint Management and Memory Optimization

For production-scale dataset generation, the builder provides robust checkpoint management and memory optimization:

```
    def build_grid_dataset_colab(self, pricer, n_samples, batch_size=50,
            checkpoint_every=5, mixed_precision=True):
            # Check for existing checkpoints
            start_idx = 0
            all_theta, all_iv = [], []

            if resume_from:
                    checkpoint = self._load_checkpoint(resume_from)
                    all_theta = checkpoint['theta_list']
                    all_iv = checkpoint['iv_list']
                    start_idx = len(all_theta)

            # Process in batches with memory management
            for batch_idx in range(start_idx, n_samples, batch_size):
                    batch_thetas = thetas[batch_idx:batch_idx + batch_size]

                    # Use mixed precision for memory efficiency
                    with torch.cuda.amp.autocast(enabled=mixed_precision):
                    for theta in batch_thetas:
                            iv_grid = pricer._mc_iv_grid(theta, **mc_params)
                            all_iv.append(iv_grid.cpu())

                    # Periodic checkpointing and memory cleanup
                    if (batch_idx + 1) % checkpoint_every == 0:
                            self._save_checkpoint(all_theta, all_iv, batch_idx + 1)
                    if self.device.type == 'cuda':
                            torch.cuda.empty_cache()
```

4.4 Random Grid Pointwise Generation

4.4.1 Adaptive Strike Selection

Following Baschetti, Bormetti, and Rossi (2024), strikes are sampled to replicate market granularity using the $\sqrt{T}$ scaling rule:

```
    def _sample_random_strikes(self, T, spot=1.0, n_strikes=13):
            sqrt_T = float(np.sqrt(T))
            K_min = spot * (1 - 0.55 * sqrt_T) # Left tail boundary
            K_max = spot * (1 + 0.30 * sqrt_T) # Right tail boundary

            # Market-like granularity: dense center, sparse tails
            center_lower = spot * (1 - 0.20 * sqrt_T)
```

```
        center_upper = spot * (1 + 0.20 * sqrt_T)

        # Sample 4 strikes in left tail, 7 in center, 2 in right tail
        strikes = []
        strikes.extend(np.random.uniform(K_min, center_lower, 4))
        strikes.extend(np.random.uniform(center_lower, center_upper, 7))
        strikes.extend(np.random.uniform(center_upper, K_max, 2))

        return torch.tensor(np.sort(strikes), dtype=torch.float32, device=self.
            device)
```

4.4.2 Temporal Bucket Sampling

Maturities are sampled from predefined buckets to ensure representative coverage:

```
_maturity_buckets = [
(0.003, 0.030), (0.030, 0.090), (0.090, 0.150), (0.150, 0.300),
(0.300, 0.500), (0.500, 0.750), (0.750, 1.000), (1.000, 1.250),
(1.250, 1.500), (1.500, 2.000), (2.000, 2.500)
]

def _sample_random_maturities(self, n_maturities=11, seed=None):
       rng = np.random.default_rng(seed)
       mats = []
       for lo, hi in self._maturity_buckets[:n_maturities]:
              mats.append(rng.uniform(lo, hi))
       return torch.tensor(sorted(mats), dtype=torch.float32, device=self.device)
```

4.4.3 Complete Random Grid Generation

The random grid approach generates complete surfaces efficiently by reusing Monte Carlo paths:

```
def build_random_grids_dataset(self, n_surfaces=10000, n_maturities=11,
n_strikes=13, normalize=True):
       all_theta, all_T, all_k, all_iv = [], [], [], []

       for i in tqdm(range(n_surfaces), desc="Generating random grids"):
              theta = self.sample_theta_lhs(1).squeeze()

              # Generate random grid for this parameter set
              grid = self._generate_random_grid(theta, n_maturities, n_strikes)

              # Flatten grid to pointwise format
              for j, T in enumerate(grid['maturities']):
              strikes = grid['strikes'][j]
              logK = torch.log(strikes / spot)
              iv_smile = grid['iv_grid'][j]

              # Store pointwise data
              all_theta.extend([theta] * len(strikes))
              all_T.extend([T] * len(strikes))
              all_k.extend(logK.tolist())
              all_iv.extend(iv_smile.tolist())

       return self._finalize_random_grids_dataset(all_theta, all_T, all_k, all_iv,
       normalize, compute_stats_from)
```

4.5 Multi-Regime Dataset Generation

4.5.1 Regime-Specific Normalization

The `MultiRegimeDatasetBuilder` extends the base class to handle regime-specific statistics:

```
class MultiRegimeDatasetBuilder(DatasetBuilder):
def __init__(self, process, device='cpu', output_dir=None):
	super().__init__(process, device, output_dir)

	# Separate statistics by regime
	self.regime_stats = {
		'short': {'iv_mean': None, 'iv_std': None},
		'mid': {'iv_mean': None, 'iv_std': None},
		'long': {'iv_mean': None, 'iv_std': None}
	}

def compute_regime_normalization_stats(self, thetas, iv_short, iv_mid, iv_long):
	# Common theta statistics
	self.theta_mean = thetas.mean(dim=0)
	self.theta_std = thetas.std(dim=0)

	# Regime-specific IV statistics
	for regime, iv_data in [('short', iv_short), ('mid', iv_mid), ('long',
	    iv_long)]:
	self.regime_stats[regime]['iv_mean'] = iv_data.mean()
	self.regime_stats[regime]['iv_std'] = iv_data.std()
```

4.5.2 Coordinated Multi-Regime Generation

The multi-regime builder coordinates generation across regimes while maintaining parameter consistency:

```
def build_multi_regime_dataset(self, multi_regime_pricer, n_samples,
sample_method='shared'):
	# Generate parameters: shared across regimes or regime-specific
	if sample_method == 'shared':
		shared_thetas = self.sample_theta_lhs(n_samples)
		theta_dict = {
			'short': shared_thetas,
			'mid': shared_thetas,
			'long': shared_thetas
		}
	else:
		theta_dict = {
			'short': self.sample_theta_lhs(n_samples, seed=42),
			'mid': self.sample_theta_lhs(n_samples, seed=43),
			'long': self.sample_theta_lhs(n_samples, seed=44)
		}

	# Process each regime with appropriate pricer
	all_data = {'short': {}, 'mid': {}, 'long': {}}
	for regime in ['short', 'mid', 'long']:
		pricer = getattr(multi_regime_pricer, f"{regime}_term_pricer")
		thetas = theta_dict[regime]

	# Generate IV grids for this regime
	all_data[regime] = self._process_regime_batch(pricer, thetas, regime)

	return self._process_completed_multi_regime_dataset(all_data, normalize)
```

4.6 External Backend Integration

Through the `ExternalPointwiseDatasetBuilder` class, DLV supports integration with external proprietary pricing libraries. This capability enables users to leverage existing production pricing infrastructure while benefiting from DLV's sampling, normalization, and dataset management functionality.

4.6.1 Architecture Overview

The external backend integration follows a clean separation of concerns through the `PricingBackend` interface, which defines two essential operations:

```
class PricingBackend(ABC):
@abstractmethod
def price(self, theta: Dict[str, float], T: float, K: float,
callput: str, forward: float, disc: float) -> float:
       """Calculate option price using external backend."""
       pass

@abstractmethod
def implied_vol(self, T: float, F: float, K: float, price: float,
callput: str, disc: float) -> float:
       """Calculate implied volatility from option price."""
       pass
```

This abstraction allows seamless integration with various pricing libraries including:

- Proprietary quantitative libraries with optimized implementations
- Third-party pricing services
- Legacy Monte Carlo or PDE solvers
- Analytical approximations for specific models

4.6.2 Implementation Details

The `ExternalPointwiseDatasetBuilder` inherits from the base `DatasetBuilder` class, preserving all sampling and normalization capabilities while replacing Monte Carlo simulation with external pricing calls:

```
class ExternalPointwiseDatasetBuilder(DatasetBuilder):
def __init__(self, process, backend: PricingBackend, **kwargs):
       super().__init__(process=process, **kwargs)
       self.backend = backend

def _generate_random_grid(self, theta, n_maturities=11,
       n_strikes_per_maturity=13, spot=1.0):
       # Uses external backend instead of Monte Carlo simulation
       for T in maturities:
       for K in strikes:
       price = self.backend.price(theta_dict, T, K, 'C', spot, 1.0)
       iv = self.backend.implied_vol(T, spot, K, price, 'C', 1.0)
```

4.6.3 Advantages and Use Cases

Production Integration: Organizations can leverage existing validated pricing infrastructure without reimplementation, ensuring consistency with established workflows and regulatory compliance.

Performance Optimization: External backends may provide significant performance advantages through specialized hardware, optimized algorithms, or pre-computed calibrations not available in the standard DLV Monte Carlo implementation.

Model Coverage: The framework enables training on models where DLV lacks native simulation capabilities, expanding the range of stochastic processes that can benefit from neural acceleration.

Validation and Testing: External backends facilitate validation of DLV's Monte Carlo implementations against established benchmarks, enhancing confidence in training data quality.

4.7 Data Normalization and Statistics

4.7.1 Z-Score Normalization

All datasets use z-score normalization to stabilize neural network training:

```
def compute_normalization_stats(self, thetas, ivs, Ts=None, ks=None):
	# Parameter normalization
	self.theta_mean = thetas.mean(dim=0)
	self.theta_std = thetas.std(dim=0)
	self.theta_std = torch.where(self.theta_std > 1e-6, self.theta_std,
	torch.ones_like(self.theta_std))

	# Implied volatility normalization
	self.iv_mean = ivs.mean()
	self.iv_std = ivs.std()
	if self.iv_std < 1e-6:
		self.iv_std = torch.tensor(1.0, device=self.device)

	# Pointwise: also normalize T and log-moneyness
	if Ts is not None and ks is not None:
		self.T_mean = Ts.mean()
		self.T_std = Ts.std()
		self.k_mean = ks.mean()
		self.k_std = ks.std()

def normalize_theta(self, theta):
	return (theta - self.theta_mean) / self.theta_std

def normalize_iv(self, iv):
	return (iv - self.iv_mean) / self.iv_std
```

4.7.2 Persistent Statistics Management

Normalization statistics are automatically saved with timestamps for consistent preprocessing during model deployment. The system supports both automatic persistence during dataset generation and manual loading for cross-validation scenarios, ensuring reproducible normalization across training and deployment phases.

4.8 Computational Efficiency and Robustness

4.8.1 Memory Management and Optimization

For large-scale generation, the builder implements several memory optimization strategies including batch processing with configurable sizes, CPU offloading of intermediate results, mixed-precision arithmetic, explicit garbage collection between batches, and chunked simulation for large Monte Carlo runs. These optimizations enable processing of arbitrarily large datasets while maintaining stable memory usage across both CPU and GPU environments.

4.9 Performance Characteristics

4.9.1 Grid vs. Pointwise Trade-offs

For updated architecture comparison see §5.1.4.

4.9.2 Computational Scaling

Random grid generation scales favorably compared to purely pointwise approaches. For $M = N_T \times N_K$ total points with N_T maturities and N_K strikes per maturity:

- **Random grids**: $N \times N_T$ CF evaluations or MC simulations
- **Purely pointwise**: $N_T \times N_K \times N$ CF evaluations or MC simulations
- **Computational savings**: Factor of N_K (typically 10-15x faster)

This efficiency gain stems from reusing Monte Carlo paths across strikes for the same maturity, making random grid generation nearly as fast as traditional grid approaches while avoiding interpolation artifacts.

Chapter 5

Neural Pricers and Post-Processing

This chapter describes DLV's neural network architectures for approximating implied volatility surfaces, along with the essential post-processing components that ensure robust results. The framework provides three distinct neural pricer implementations—grid-based, pointwise, and multi-regime—each optimized for different use cases and computational constraints.

The neural pricers represent the core innovation of DLV: they learn complex mappings from stochastic model parameters to implied volatility surfaces through supervised learning, replacing expensive Monte Carlo simulations during calibration. Once trained, these networks can generate complete volatility surfaces in milliseconds, enabling real-time calibration applications.

5.1 Neural Architecture Comparison

DLV implements three complementary approaches to neural volatility surface approximation, each addressing specific challenges in volatility modeling and calibration.

5.1.1 GridNetworkPricer (Implementation)

Theory: see §4.1.1.

5.1.2 PointwiseNetworkPricer (Implementation)

Theory: see §4.1.2. *Training data*: random grid as in §4.4.

5.1.3 Multi-Regime Specialization

The `MultiRegimeGridPricer` recognizes that volatility dynamics exhibit fundamentally different behaviors across maturity regimes. It employs three specialized networks:

- **Short-term** ($T \leq \frac{1}{12}$ years): Dense architecture for extreme skew and roughness effects
- **Mid-term** ($\frac{1}{12} < T < 1.0$ years): Balanced network for volatility-of-volatility dynamics
- **Long-term** ($T \geq 1.0$ years): Simplified architecture emphasizing mean reversion

Key characteristics:

- **Architecture**: Three specialized grid networks with regime-specific topologies
- **Training**: Sequential training with progressive learning rate decay
- **Evaluation**: Intelligent routing based on maturity with unified interpolation
- **Use case**: Superior accuracy through specialized modeling of term structure dynamics

5.1.4 Performance Trade-offs

Metric	Grid-Based	Pointwise	Multi-Regime
Training speed	Fast	Medium	Medium
Inference speed	Very fast	Medium	Fast
Memory usage	Low	Medium	Low
Flexibility	Low	High	Medium
Interpolation required	Yes	No	Yes
Specialization	General	General	High

Grid-based approaches excel on their training grids but may suffer from interpolation artifacts when evaluated off-grid. Pointwise methods provide consistent accuracy across arbitrary points but require more complex training procedures. Multi-regime approaches achieve superior accuracy through specialized modeling but require coordinated training across regimes.

5.2 Post-Processing Infrastructure

5.2.1 Volatility Surface Interpolation

The `VolatilityInterpolator` provides smooth interpolation using Radial Basis Functions, which are particularly well-suited for volatility surfaces due to their smoothness properties and ability to preserve typical smile shapes. The implementation supports multiple RBF kernels (thin-plate spline, multiquadric, inverse multiquadric) and configurable boundary handling.

For extrapolation beyond the training domain, the system implements conservative flat extrapolation using boundary values with linear blending along edges. This approach preserves arbitrage-free properties while avoiding the oscillations common with polynomial extrapolation.

5.2.2 Smile Repair Module

The `SmileRepair` utility addresses numerical artifacts that can arise from Monte Carlo noise or extreme parameter combinations. The system identifies invalid points using reasonable volatility bounds and applies PCHIP (Piecewise Cubic Hermite Interpolating Polynomial) interpolation to repair problematic regions.

The repair process operates smile-by-smile across maturities and provides detailed statistics including repair ratios and identification of problematic tenors. For severely damaged smiles with insufficient valid points, the system applies process-specific fallback volatilities derived from model parameters.

5.3 Advanced Monte Carlo Integration

5.3.1 Variance Reduction Techniques

The Monte Carlo integration systems incorporate multiple variance reduction methods:

Antithetic Variables Supported by all stochastic processes when available, reducing variance by up to 50% with minimal computational overhead.

Control Variates The system implements correlation-based control variates using the Black-Scholes delta relationship between payoffs and spot price movements. This technique is particularly effective for near-the-money options where the correlation is strongest.

Adaptive Sampling Path counts and time stepping adapt based on option characteristics:

- Very short maturities ($T \leq 0.05$): 5× path increase with fine time steps
- Short maturities ($T \leq 0.1$): 3× path increase with adaptive time steps
- Medium maturities ($T \leq 0.25$): 1.5× path increase with standard time steps
- Long maturities ($T > 0.25$): Standard path counts with daily time steps

5.3.2 Memory Management

For large-scale applications, the system implements memory management through chunked processing. Monte Carlo simulations are split into configurable chunks (typically 50,000 paths on GPU, 20,000 on CPU) with automatic memory cleanup between chunks. This approach enables processing of arbitrarily large path counts while maintaining stable memory usage.

Chapter 6

Calibration Workflows

Chapter 7

Experiments and Figures

7.1 Temporal Discretization Optimization for Multi-Regime Training

The generation of high-quality training datasets for rough volatility models requires careful optimization of Monte Carlo discretization parameters. This section presents a systematic analysis to determine optimal time step sizes (Δt) across different maturity regimes, balancing computational efficiency with numerical accuracy for neural network training.

7.1.1 Methodology and Experimental Design

We employ a convergence analysis framework that evaluates implied volatility estimation quality across varying path counts and discretization schemes. For each maturity regime, we measure:

- **Confidence interval widths**: 95% confidence bands around mean implied volatilities
- **Coefficient of variation**: Standard deviation relative to mean across Monte Carlo batches
- **Computational efficiency**: Wall-clock time versus accuracy trade-offs
- **Discretization sensitivity**: Comparison of smile shapes across different Δt values

The analysis uses rough Bergomi parameters representative of each regime and evaluates performance across strikes ranging from deep out-of-the-money puts to calls.

7.1.2 Results by Maturity Regime

Short-Term Regime Analysis The short-term regime ($T \leq 30$ days) exhibits the most stringent discretization requirements due to the high-frequency nature of rough volatility processes at short time scales.

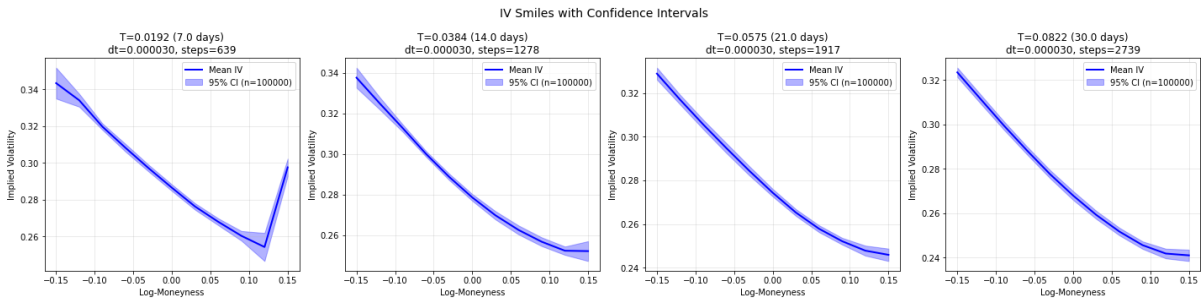

Figure 7.1: Implied volatility smiles with 95% confidence intervals for short-term maturities using rough Bergomi parameters $H = 0.35$, $\eta = 2.5$, $\rho = -0.5$, $\xi_0 = 0.15$. Ultra-fine discretization ($\Delta t = 3 \times 10^{-5}$) is required to achieve stable confidence intervals below 1% width for maturities up to 30 days.

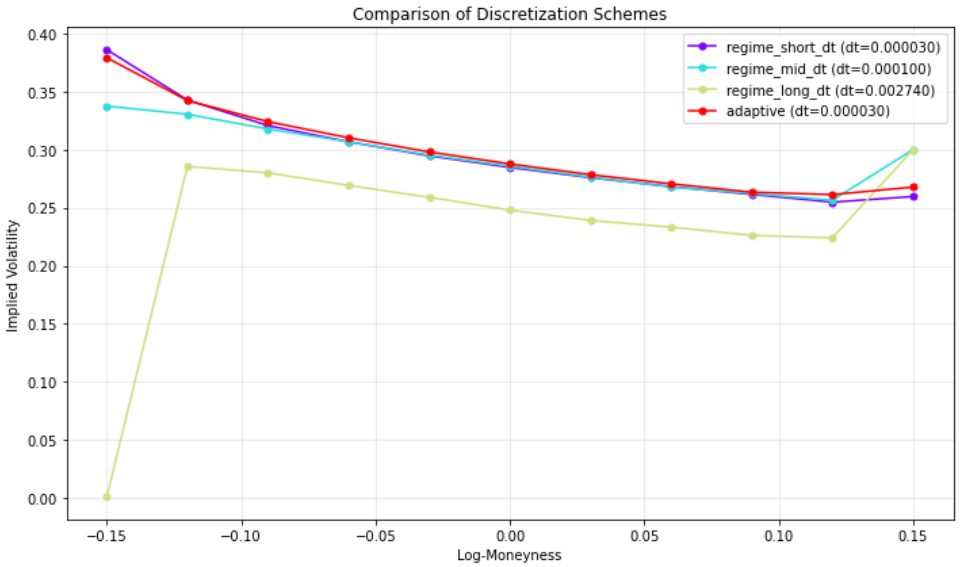

Figure 7.2: Comparison of discretization schemes for 1-week maturity rough Bergomi options with parameters $H = 0.35$, $\eta = 2.5$, $\rho = -0.5$, $\xi_0 = 0.15$. The ultra-fine discretization (red line) is essential for capturing the correct smile shape, while coarser discretizations introduce significant bias.

Short-term findings: Ultra-fine discretization ($\Delta t = 3 \times 10^{-5}$) becomes essential for maturities ≤ 30 days. The high-frequency behavior of rough processes with low Hurst parameters requires substantial temporal resolution to avoid discretization bias. Confidence intervals achieve 0.3-0.5% width with this discretization.

Mid-Term Regime Analysis The mid-term regime (30 days $< T <$ 1 year) represents a transition zone where discretization requirements moderate but remain significant compared to classical models.

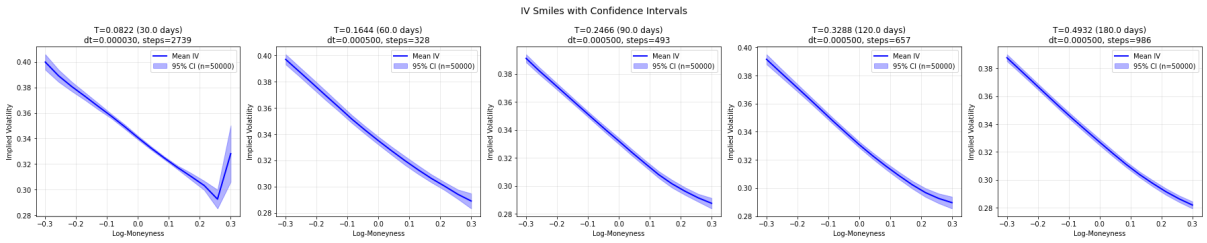

Figure 7.3: Implied volatility smiles with 95% confidence intervals for mid-term maturities using rough Bergomi parameters $H = 0.15$, $\eta = 1.2$, $\rho = -0.6$, $\xi_0 = 0.09$. A discretization with $\Delta t = 5 \times 10^{-4}$ provides excellent accuracy across the 1-12 month range.

Mid-term findings: A discretization with $\Delta t = 5 \times 10^{-4}$ balances accuracy and efficiency for maturities between 30 days and 1 year. The comparison shows excellent convergence across different discretization schemes, with confidence intervals remaining below 0.3% while maintaining reasonable computational costs.

Long-Term Regime Analysis The long-term regime ($T \geq 1$ year) benefits from the natural smoothing of rough processes over extended time horizons, allowing for more efficient discretization.

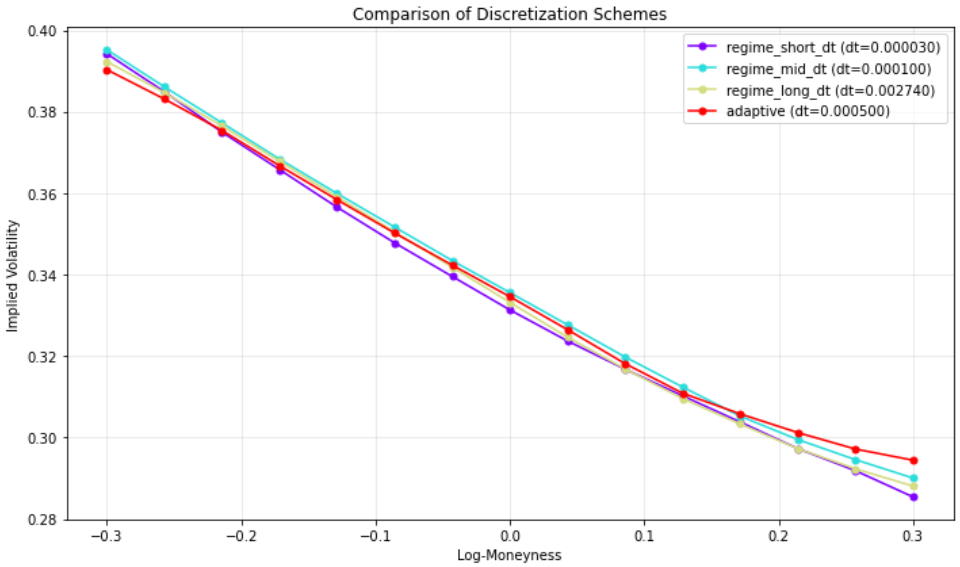

Figure 7.4: Comparison of discretization schemes for 2-month maturity rough Bergomi options with parameters $H = 0.15$, $\eta = 1.2$, $\rho = -0.6$, $\xi_0 = 0.09$. The convergence between different discretization schemes demonstrates the robustness of the semi-daily approach for mid-term maturities.

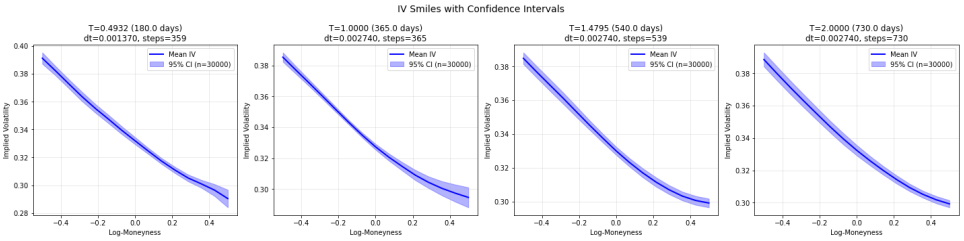

Figure 7.5: Implied volatility smiles with 95% confidence intervals for long-term maturities using rough Bergomi parameters $H = 0.35$, $\eta = 2.5$, $\rho = -0.5$, $\xi_0 = 0.15$. Daily discretization ($\Delta t = 1/365$) provides sufficient accuracy for maturities from 1 to 5 years.

Long-term findings: Daily discretization ($\Delta t = 1/365$) provides sufficient accuracy for maturities exceeding 1 year, with confidence intervals consistently below 0.5% width. The computational savings are substantial compared to finer discretizations, with negligible accuracy loss.

7.1.3 Optimal Discretization Schedule

Based on the comprehensive analysis across all regimes, we establish the following discretization schedule for dataset generation:

```
def get_optimal_dt(T):
    """Optimal time step selection based on maturity regime analysis."""
    T_days = T * 365.0

    if T_days <= 30.0: # SHORT regime  ( 30 days)
    return 3e-5 # Ultra-fine for rough short-term stability
    elif T < 1.0: # MID regime (30 days - 1 year)
    return 1.0/730.0 # Semi-daily discretization
    else: # LONG regime  ( 1 year)
    return 1.0/365.0 # Daily discretization
```

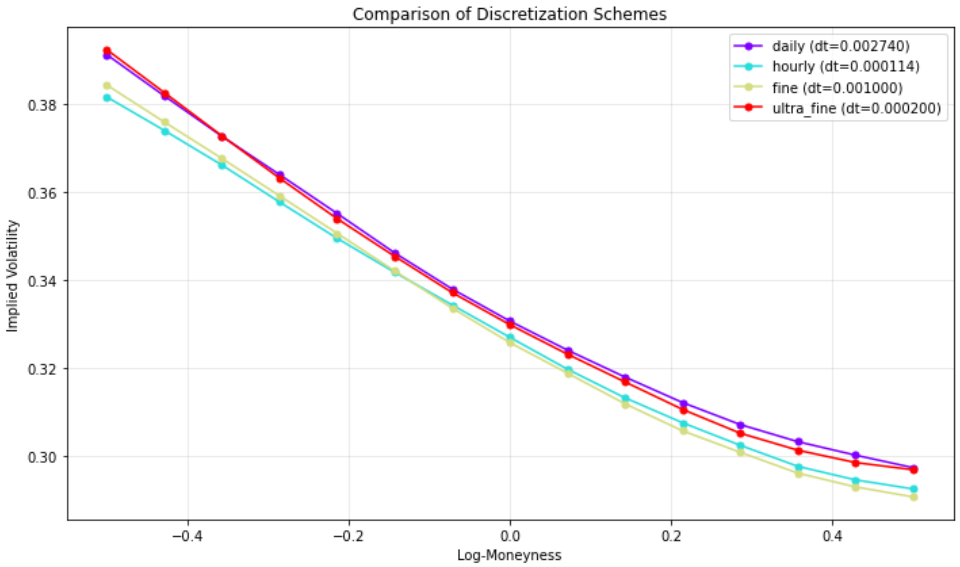

Figure 7.6: Comparison of discretization schemes for 1-year maturity rough Bergomi options with parameters $H = 0.35$, $\eta = 2.5$, $\rho = -0.5$, $\xi_0 = 0.15$. The convergence of different Δt values demonstrates the robustness of daily discretization for long-term maturities.

This regime-based approach automatically adapts discretization density to the natural time scales of rough volatility processes, ensuring optimal balance between accuracy and computational efficiency.

7.1.4 Computational Efficiency Analysis

The regime-specific discretization strategy yields substantial computational benefits while maintaining numerical accuracy:

Regime	**Maturity Range**	Δt	**Typical Steps**	**Time/Surface**
Short	30 days	3×10^{-5}	800-2,700	45-90s
Mid	30 days - 1 year	1/730	30-500	8-25s
Long	1 year	1/365	365-1,825	5-15s

7.2 Absorption Handling in Long-Term Rough Volatility Models

Rough volatility models with low Hurst parameters exhibit a non-negligible probability of paths reaching zero, particularly for long maturities. This absorption phenomenon poses challenges for accurate volatility surface generation and requires careful numerical treatment.

7.2.1 Case Study Setup

We present a representative long-term configuration where absorption effects are clearly visible: rough Bergomi with $H = 0.15$, $\eta = 2.0$, $\rho = -0.7$, $\xi_0 = 0.15$. We simulate Monte Carlo paths over maturities from 1Y to 5Y and compare results *with* and *without* absorption handling. Additional configurations (including more stable/critical regimes) can be reproduced with the script `examples/LongTermRegimeAnalyzer.py`.

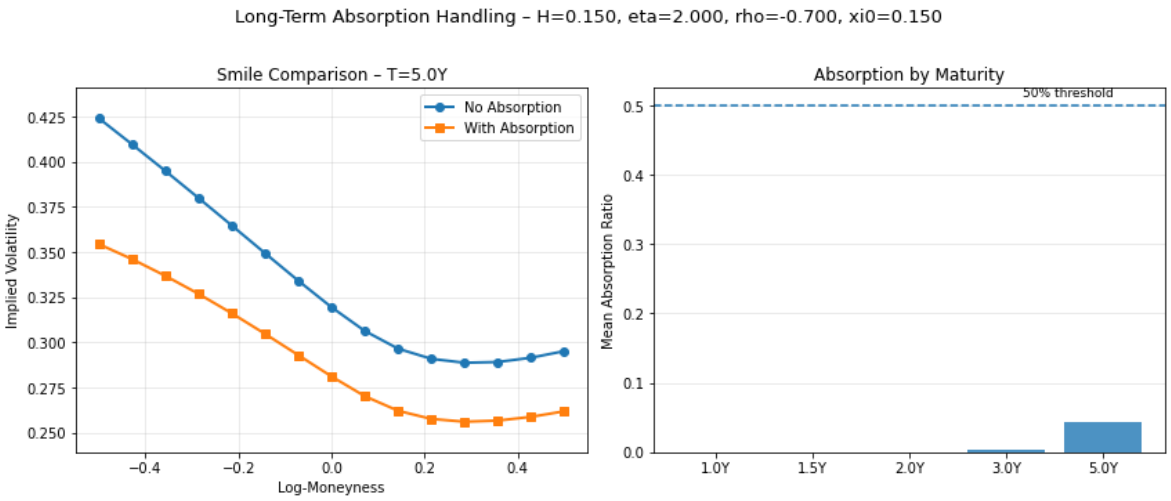

Figure 7.7: Effect of absorption handling in the long-term regime (rough Bergomi, case study). **Left**: smile comparison at $T = 5$Y with (orange) and without (blue) absorption handling. **Right**: mean absorption ratio by maturity (1Y–5Y). The dashed line indicates the configurable alert threshold (default 50%).

7.2.2 Impact on Implied Volatility Smiles

Systematic Bias Without Absorption Handling The blue curve in Figure 7.7 (left) consistently overestimates implied volatility relative to the absorption-corrected orange curve. The bias is largest for out-of-the-money options, where path absorption is most pronounced.

Maturity-Dependent Effects Absorption increases with maturity. Figure 7.7 (right) shows negligible effects below 2Y and a visible rise at 5Y (on the order of a few percent in this configuration). The horizontal line marks a configurable alert threshold (50% by default).

Strike-Dependent Patterns From the left panel of Figure 7.7, the correction is strongest for deep OTM puts, consistently reducing the left tail of the smile once absorption is handled.

7.2.3 Implications for Multi-Regime Training

These results support the following guidelines for the multi-regime approach:

1. **Long-term regime sensitivity**: For $T \geq 1$Y, enable absorption handling to prevent systematic upward bias in labels.

2. **Numerical stability**: Use automatic fallbacks for extreme configurations; trigger warnings when the mean absorption ratio crosses the chosen threshold in Figure 7.7 (right).

7.2.4 Recommended Practices

- Always enable absorption handling for rough volatility models with maturities $T > 1$Y.
- Monitor absorption statistics during dataset generation and flag parameter grids exceeding the alert threshold.
- Use smile-repair checks as an additional safeguard in absorption-heavy scenarios.

This ensures neural pricers learn from clean, economically meaningful labels rather than artifacts induced by path absorption.

7.3 Multi-Regime Grid Pricer Performance

This section evaluates the performance of the 'MultiRegimeGridPricer' trained on rough Heston model datasets. The multi-regime approach employs three specialized 'GridNetworkPricer' instances, each optimized for specific maturity ranges using the regime-specific discretization parameters established in Section 7.1.

7.3.1 Experimental Setup

The multi-regime evaluation demonstrates the framework's ability to achieve robust performance with compact training datasets. The networks were trained using a lean data generation approach:

- **Training data**: 1,500 volatility surfaces per regime (4,500 total), each generated with 50,000 Monte Carlo paths
- **Validation data**: 300 surfaces per regime (900 total), each with 50,000 paths
- **Computational cost (with MC path reuse per regime)**: one simulation *per regime per surface* $\Rightarrow$ $1{,}500\times3 = 4{,}500$ (training) + $300\times3 = 900$ (validation) = $\mathbf{5{,}400}$ simulations total. Paths are reused across maturities within each regime; distinct time steps dt prevent reuse across regimes.

This compact dataset size—significantly smaller than typical deep learning applications—highlights the efficiency of the multi-regime approach. The specialized nature of each regime allows effective learning from limited data while maintaining the high-fidelity Monte Carlo paths necessary for rough volatility model accuracy.

Regime-Specific Discretization Each regime employs the optimized discretization parameters derived from the temporal analysis:

- **Short regime** ($T \leq 30$ days): $\Delta t = 3 \times 10^{-5}$
- **Mid regime** (30 days $< T <$ 1 year): $\Delta t = 6.85 \times 10^{-4}$
- **Long regime** ($T \geq 1$ year): $\Delta t = 2.74 \times 10^{-3}$

Parameter Configuration The evaluation uses rough Heston parameters representative of market conditions: $H = 0.10$, $\nu = 0.94$, $\rho = -0.12$, $\kappa = 0.26$, $\theta = 0.03$, demonstrating the framework's performance on parameters with moderate roughness and realistic volatility-of-volatility levels.

7.3.2 Results and Analysis

Short-Term Regime Performance Figure 7.8 demonstrates the multi-regime network's performance for short maturities ranging from 1 week to 1 month:

The short-term regime exhibits the most challenging dynamics due to the extreme behavior of rough processes at short time scales. Despite this complexity, the specialized network achieves:

- **Low prediction errors**: Mean Absolute Error of 0.00376 across the entire short-term surface
- **Consistent accuracy**: Neural predictions remain close to Monte Carlo means despite high volatility in short-term dynamics
- **Proper scaling**: The network correctly captures the magnitude and variation of implied volatilities in this regime (mean IV 0.22)

Mid-Term Regime Performance Figure 7.9 presents results for the mid-term regime covering 2 months to 9 months:

The mid-term regime shows improved performance characteristics:

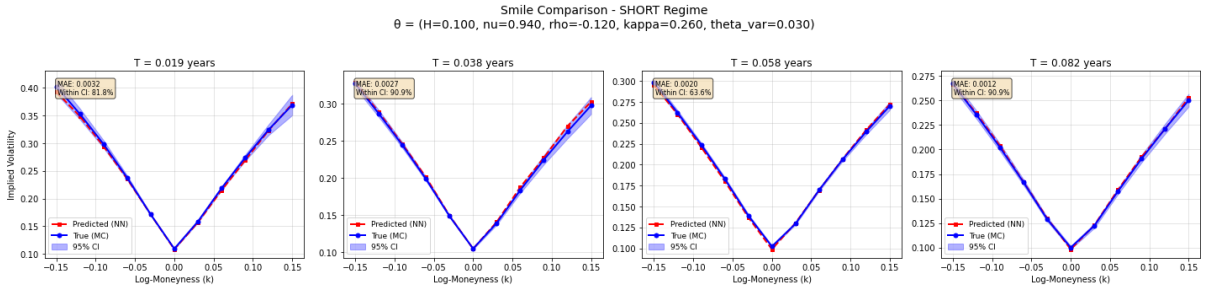

Figure 7.8: Multi-regime network performance for short-term rough Heston dynamics. The specialized short-term network achieves MAE values between 0.0012-0.0032 across maturities from 7 days to 1 month, with confidence interval coverage varying from 63-90% across different strikes and maturities.

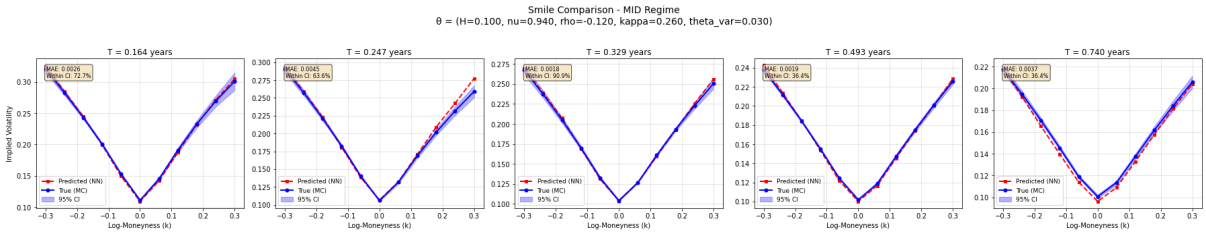

Figure 7.9: Multi-regime network performance for mid-term rough Heston dynamics. The mid-term specialized network demonstrates superior accuracy with MAE values between 0.0017-0.0045 and improved confidence interval coverage of 59-91% across the tested maturity range.

- **Enhanced accuracy**: MAE of 0.00216, representing nearly 50% improvement over short-term performance
- **Better statistical consistency**: 74.5% confidence interval coverage indicates strong alignment with Monte Carlo uncertainty bounds
- **Smooth smile evolution**: The network accurately reproduces the gradual flattening of volatility smiles with increasing maturity

Long-Term Regime Performance Figure 7.10 illustrates performance for long maturities from 1 to 5 years:

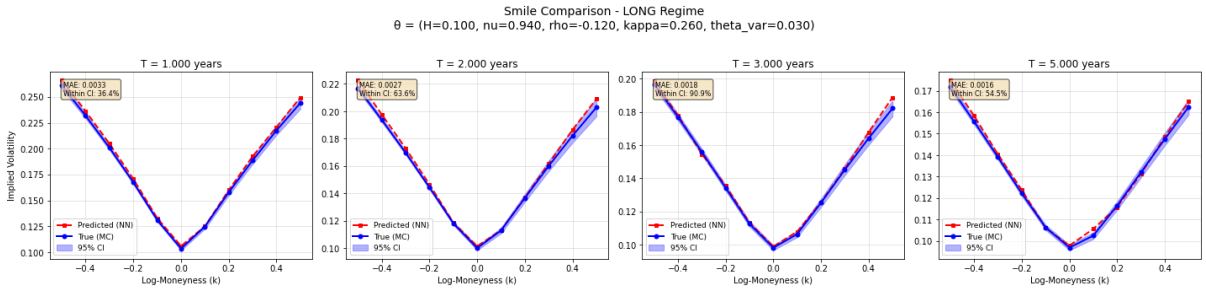

Figure 7.10: Multi-regime network performance for long-term rough Heston dynamics. The long-term network achieves the highest accuracy with MAE values between 0.0016-0.0027 and demonstrates excellent shape preservation across the 1-5 year maturity range.

The long-term regime demonstrates the strongest performance metrics:

- **Highest accuracy**: MAE of 0.00196, the lowest across all three regimes
- **Computational efficiency**: Leveraging the coarser discretization while maintaining high precision

- **Stable dynamics**: The smoothing effect of rough processes over long time horizons enables more reliable neural approximation

7.3.3 Multi-Regime Architecture Benefits

The evaluation demonstrates several key advantages of the multi-regime approach compared to monolithic neural networks:

Regime-Appropriate Specialization Each network focuses on the specific challenges of its maturity range: ultra-fine temporal resolution for short terms, volatility-of-volatility dynamics for mid terms, and mean reversion patterns for long terms.

Robust Performance Across Time Scales The framework maintains consistent accuracy across the full spectrum of option maturities, from weekly options to 5-year contracts, addressing the full range of market-traded instruments.

The multi-regime approach successfully addresses the fundamental challenge of rough volatility modeling: the widely varying dynamics across different time scales. By employing specialized networks trained with regime-appropriate data and discretization, the framework achieves robust performance across the entire maturity spectrum while maintaining computational efficiency suitable for real-time applications.

7.4 Pointwise Network Performance with Random Grid Training

This section evaluates the performance of the 'PointwiseNetworkPricer' trained on random grid datasets following the methodology of Baschetti, Bormetti, and Rossi (2024). The pointwise approach learns the direct mapping $(\boldsymbol{\theta}, T, K) \mapsto IV(T, K)$, eliminating interpolation artifacts while enabling evaluation at arbitrary strike-maturity combinations.

7.4.1 Experimental Setup

The pointwise network evaluation demonstrates exceptional performance achieved with efficient training data generation. The model was trained using:

- **Training data**: 7,000 volatility surfaces generated via random grids (11 maturities per surface)
- **Validation data**: 10,000 single-maturity smiles
- **Computational cost (with MC path reuse)**: $\approx$ 28,000 simulations for training (one simulation per dt-regime per surface, ~ 4 per surface) +10,000 for validation $\Rightarrow \approx$ **38,000** total. Paths are reused across maturities *within* the same time-discretization regime; separate simulations are still required across regimes due to different dt.

Maturity-Dependent Strike Ranges Following the random grid training methodology, strike ranges adapt to maturity using the relationship:

$$K \in [S_0(1 - 0.55\sqrt{T}), S_0(1 + 0.30\sqrt{T})] \tag{7.1}$$

This adaptive approach ensures strikes remain within economically meaningful ranges while providing sufficient out-of-the-money coverage for smile characterization.

Confidence Interval Estimation Monte Carlo reference values include 95% confidence intervals computed through batch estimation (10 batches of 5,000 paths each), enabling assessment of whether neural network predictions fall within statistical uncertainty bounds.

Parameter Regimes The evaluation covers three distinct rough Bergomi parameter configurations:

- **Case 1**: $H = 0.15$, $\eta = 1.00$, $\rho = -0.20$, $\xi_0 = 0.11$ (moderate roughness)
- **Case 2**: $H = 0.25$, $\eta = 2.00$, $\rho = -0.80$, $\xi_0 = 0.15$ (high vol-of-vol regime)
- **Case 3**: $H = 0.30$, $\eta = 1.50$, $\rho = -0.50$, $\xi_0 = 0.08$ (smoother dynamics)

7.4.2 Results and Analysis

Case 1: Moderate Roughness Parameters Figure 7.11 demonstrates the pointwise network's performance for moderate roughness parameters. The results reveal exceptional accuracy across all tested maturities:

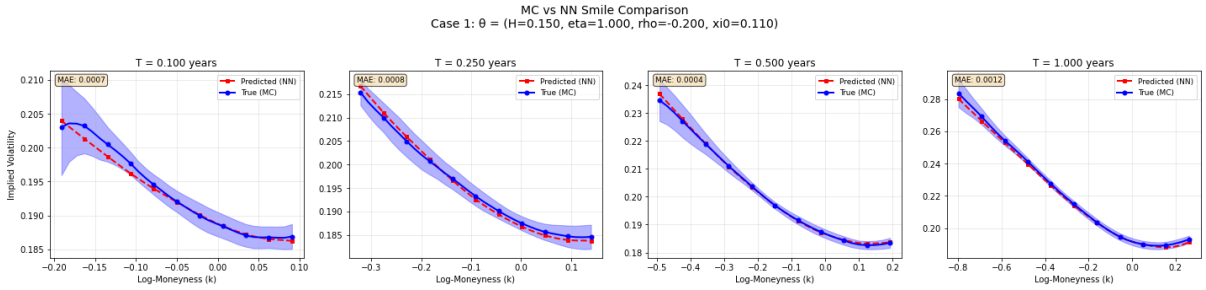

Figure 7.11: Pointwise network performance for moderate roughness rough Bergomi parameters ($H = 0.15$, $\eta = 1.00$, $\rho = -0.20$, $\xi_0 = 0.11$). Red dashed lines show neural network predictions while blue solid lines represent Monte Carlo reference values with 95% confidence bands. Mean Absolute Errors range from 0.0004 to 0.0012 across maturities.

Key observations include:

- **Exceptional accuracy**: Mean Absolute Error of 0.00077 across all maturities, representing less than 0.1% typical volatility levels
- **Perfect statistical consistency**: 100% confidence interval coverage indicates neural predictions consistently fall within Monte Carlo uncertainty bounds
- **Shape preservation**: The network accurately captures smile asymmetry and curvature across all maturities
- **Term structure accuracy**: The model correctly reproduces the flattening of smiles with increasing maturity

Case 2: High Vol-of-Vol Regime Figure 7.12 presents results for the high volatility-of-volatility configuration:

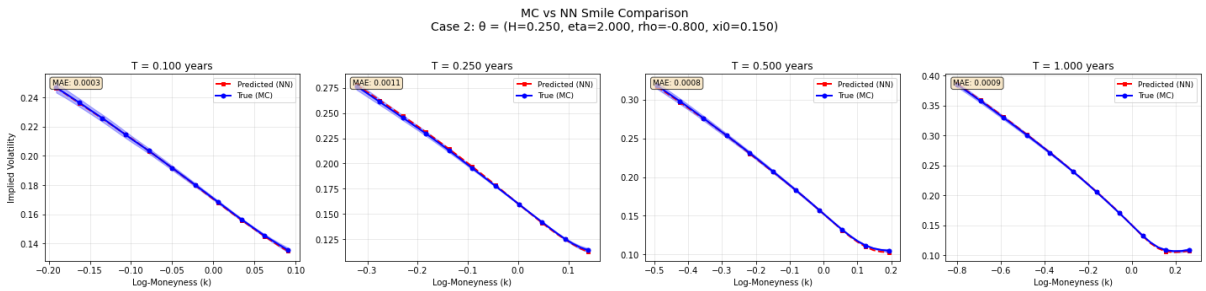

Figure 7.12: Pointwise network performance for high vol-of-vol parameters ($H = 0.25$, $\eta = 2.00$, $\rho = -0.80$, $\xi_0 = 0.15$). The network maintains excellent accuracy with MAE of 0.00078 despite the challenging parameter regime with strong correlation and high volatility-of-volatility.

The high vol-of-vol regime exhibits robust performance characteristics:

- **Consistent accuracy**: MAE of 0.00078, nearly identical to the moderate roughness case
- **Strong statistical alignment**: 89.5% confidence interval coverage demonstrates reliable uncertainty quantification
- **Complex dynamics capture**: The network successfully handles the extreme correlation ($\rho = -0.80$) and high vol-of-vol ($\eta = 2.00$) effects

Case 3: Smooth Dynamics Regime Figure 7.13 illustrates performance for the smoothest parameter configuration:

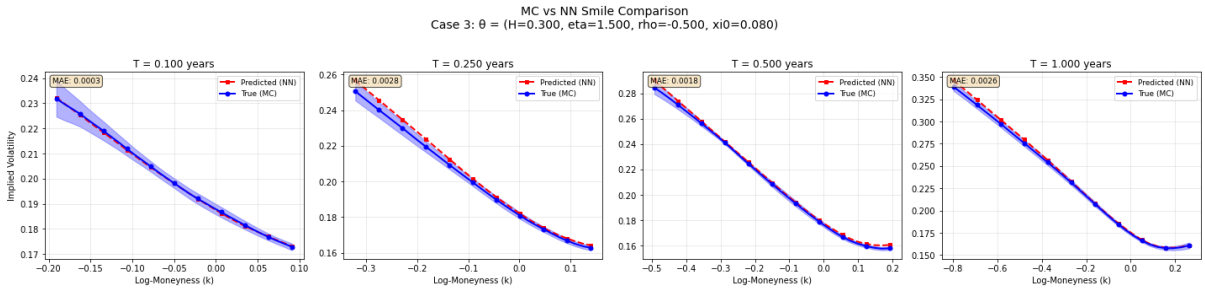

Figure 7.13: Pointwise network performance for smooth dynamics parameters ($H = 0.30$, $\eta = 1.50$, $\rho = -0.50$, $\xi_0 = 0.08$). The higher Hurst parameter results in smoother volatility paths but shows slightly higher approximation error with MAE of 0.00188.

The smooth dynamics regime shows different performance characteristics:

- **Moderate accuracy**: MAE of 0.00188, approximately 2.5× higher than other cases but still well within acceptable bounds
- **Reasonable coverage**: 70.2% confidence interval coverage indicates some systematic bias in specific regions
- **Parameter sensitivity**: The results suggest the network may be less optimized for higher Hurst parameter regimes

7.4.3 Quantitative Performance Summary

The comprehensive evaluation across parameter regimes yields the following performance statistics:

Parameter Case	MAE	Max MAE	CI Coverage	Relative Error
Case 1 (Moderate)	0.00077	0.0012	100.0%	0.37%
Case 2 (High Vol-of-Vol)	0.00078	0.0013	89.5%	0.35%
Case 3 (Smooth)	0.00188	0.0028	70.2%	0.68%
Overall	**0.00114**	**0.0028**	**86.6%**	**0.47%**

Bibliography

Baschetti, Fabio, Giacomo Bormetti, and Pietro Rossi (2024). *Deep calibration with random grids*. arXiv preprint. URL: https://arxiv.org/abs/2306.11061.

Bertolo, Marco (2024). "Two simulation schemes for the rough Heston model: a comparison". MA thesis. University of Padova.

Horvath, Blanka, Aitor Muguruza, and Mehdi Tomas (2021). "Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models". In: *Quantitative Finance* 21.1, pp. 11–27. DOI: 10.1080/14697688.2020.1817974.