PyDelt: Advancing Numerical Function Interpolation and Differentiation

Michael H. Lee

September 17, 2025

Abstract

Numerical differentiation is a fundamental technique in scientific computing, with applications ranging from physics and engineering to finance and machine learning. However, traditional approaches face challenges with noise sensitivity, accuracy limitations, and poor scaling to high-dimensional problems. This paper introduces PyDelt, a Python library that provides a comprehensive suite of interpolation-based differentiation methods with a unified interface. We compare PyDelt's capabilities with existing tools, highlighting its unique contributions: (1) a universal differentiation interface across multiple interpolation methods, (2) robust handling of noisy data through specialized algorithms, (3) comprehensive multivariate calculus support, (4) stochastic calculus extensions, and (5) seamless integration with neural network-based automatic differentiation. Performance evaluations demonstrate PyDelt's superior accuracy for noisy data and competitive computational efficiency. The library's unified API and method diversity make it particularly valuable for exploratory data analysis, algorithm comparison, and applications requiring both accuracy and noise robustness.

1 Introduction

Numerical differentiation—the approximation of derivatives from discrete data points—is a cornerstone of computational science and engineering. Applications span diverse fields including signal processing, computational physics, financial modeling, control systems, and machine learning. Despite its importance, numerical differentiation remains challenging due to inherent trade-offs between accuracy, noise sensitivity, and computational efficiency.

Traditional approaches to numerical differentiation fall into several categories:

- **Finite difference methods** approximate derivatives using discrete differences between function values at neighboring points. While conceptually simple and computationally efficient, these methods are no-

toriously sensitive to noise and suffer from accuracy limitations, particularly for higher-order derivatives [Fornberg, 1988].

- **Interpolation-based methods** fit continuous functions to discrete data points and then differentiate the resulting function analytically. These methods can provide improved noise robustness but vary widely in their accuracy and computational requirements [De Boor, 2001].
- **Automatic differentiation** computes exact derivatives by applying the chain rule to elementary operations in a computational graph. While highly accurate, these methods require access to the function definition rather than just data points [Baydin et al., 2018].
- **Symbolic differentiation** manipulates mathematical expressions directly to compute derivatives. This approach provides exact results but is limited to functions with explicit mathematical formulations [Meurer et al., 2017].

Each approach has distinct advantages and limitations, making method selection highly dependent on the specific application context. This fragmentation of methods has led to a proliferation of specialized tools, each with its own API, assumptions, and limitations.

PyDelt addresses this fragmentation by providing a unified framework that integrates multiple differentiation approaches under a consistent interface. The library emphasizes interpolation-based methods, which offer a balance between noise robustness and accuracy, while also incorporating automatic differentiation for high-dimensional problems. This paper examines PyDelt's contributions to the field of numerical differentiation and function approximation, comparing its capabilities with existing tools and highlighting its unique features.

2 Related Work

Several software libraries provide numerical differentiation capabilities, each with different approaches and strengths:

2.1 SciPy

SciPy [Virtanen et al., 2020] offers various interpolation methods through its `scipy.interpolate` module, including splines, polynomials, and radial basis functions. While these can be used for differentiation by differentiating the interpolant, the API is not specifically designed for this purpose, requiring users to manually chain interpolation and differentiation steps. SciPy's approach focuses on interpolation quality rather than derivative accuracy, with limited guidance for method selection based on derivative requirements.

2.2 NumDiffTools

NumDiffTools [NumDiffTools, 2023] specializes in numerical differentiation using adaptive finite difference methods coupled with Richardson extrapolation. It provides functions for computing derivatives, gradients, Jacobians, and Hessians with high accuracy. However, its approach is primarily based on finite differences, making it less robust to noise than interpolation-based methods. The library lacks a unified interface across different differentiation operations and does not support stochastic calculus or neural network integration.

2.3 FinDiff

FinDiff [FinDiff, 2023] implements finite difference approximations for derivatives of any order in any number of dimensions. It offers a clean API for defining differential operators and solving partial differential equations. The library excels at handling regular grids and provides accurate boundary treatment. However, like other finite difference methods, it is sensitive to noise and does not offer interpolation-based alternatives or stochastic calculus support.

2.4 JAX

JAX [JAX, 2018] provides automatic differentiation capabilities through its `jax.grad` function, allowing exact computation of derivatives for functions defined in Python. It supports forward and reverse-mode automatic differentiation, higher-order derivatives, and is highly optimized for GPU/TPU acceleration. While powerful for functions with known analytical forms, JAX cannot directly compute derivatives from discrete data points without first fitting a model. Additionally, it lacks specialized support for noisy data or stochastic processes.

2.5 SymPy

SymPy [Meurer et al., 2017] offers symbolic differentiation through its `sympy.diff` function, computing exact derivatives of mathematical expressions. This approach provides perfect accuracy but is limited to functions with explicit symbolic representations. SymPy cannot directly handle numerical data or noise, requiring users to first fit symbolic expressions to their data.

3 PyDelt: Design and Features

PyDelt was designed to address the limitations of existing tools by providing a comprehensive, unified framework for numerical differentiation and func-

tion approximation. The library's architecture is built around several core principles:

3.1 Universal Differentiation Interface

PyDelt implements a consistent `.fit().differentiate()` pattern across all interpolation methods, allowing users to easily switch between different approaches while maintaining the same code structure. This unified API significantly reduces the learning curve and facilitates method comparison.

```
# Same pattern works for all interpolators
interpolator = InterpolatorClass(**params)
interpolator.fit(input_data, output_data)
derivative_func = interpolator.differentiate(order=1, mask=None)
derivatives = derivative_func(eval_points)
```

Listing 1: Universal differentiation interface example

3.2 Multiple Interpolation Methods

PyDelt offers a diverse set of interpolation techniques, each with different strengths:

- **Local Linear Approximation (LLA)**: Uses min-normalization and linear regression within sliding windows to estimate derivatives. Particularly effective for data with varying baselines or drift [Lyness & Moler, 1966].
- **Generalized Local Linear Approximation (GLLA)**: Extends LLA to higher-order derivatives using a generalized linear approximation framework.
- **Functional Data Analysis (FDA)**: Uses spline-based smoothing with automatic parameter selection based on data characteristics [Ramsay & Silverman, 2010].
- **Spline Interpolation**: Creates smooth, continuous piecewise polynomial functions with controllable smoothing.
- **LOWESS/LOESS**: Non-parametric regression methods that fit simple models to localized subsets of data, providing robustness to outliers [Cleveland, 1979].
- **Neural Network Interpolation**: Leverages deep learning with automatic differentiation for complex functional relationships.

This diversity allows users to select the most appropriate method for their specific data characteristics and accuracy requirements.

3.3 Comprehensive Multivariate Calculus

PyDelt provides full support for multivariate calculus operations through its `multivariate` module:

- **Gradient** (∇f): For scalar functions, computes the vector of partial derivatives.
- **Jacobian** (J_f): For vector-valued functions, computes the matrix of all first-order partial derivatives.
- **Hessian** (H_f): For scalar functions, computes the matrix of all second-order partial derivatives.
- **Laplacian** ($\nabla^2 f$): For scalar functions, computes the sum of all unmixed second partial derivatives.

```
# Multivariate API pattern
mv_derivatives = MultivariateDerivatives(SplineInterpolator,
    smoothing=0.1)
mv_derivatives.fit(input_data, output_data)
gradient_func = mv_derivatives.gradient()
gradients = gradient_func(eval_points)
```

Listing 2: Multivariate calculus example

3.4 Stochastic Calculus Extensions

A unique feature of PyDelt is its support for stochastic calculus, enabling proper handling of financial derivatives and other stochastic processes:

- **Itô calculus**: Implements corrections for non-differentiable sample paths in stochastic processes.
- **Stratonovich calculus**: Provides an alternative interpretation of stochastic integrals.
- **Multiple stochastic link functions**: Supports normal, log-normal, gamma, beta, exponential, and Poisson distributions.

```
# Set stochastic link function for derivative transformations
interpolator.set_stochastic_link("lognormal", method="ito")
derivative_func = interpolator.differentiate(order=1)
```

Listing 3: Stochastic calculus example

3.5 Neural Network Integration

PyDelt integrates with deep learning frameworks (PyTorch and TensorFlow) to leverage automatic differentiation for complex functions and high-dimensional problems:

- **NeuralNetworkInterpolator**: Fits neural networks to data and uses automatic differentiation for derivatives.
- **NeuralNetworkMultivariateDerivatives**: Provides true multivariate derivatives with exact mixed partials.
- **Framework flexibility**: Supports both PyTorch and TensorFlow backends.

4 Methodology Comparison

4.1 Interpolation vs. Finite Differences

PyDelt's primary innovation is its focus on interpolation-based differentiation rather than direct finite differences. This approach offers several advantages:

- **Noise robustness**: Interpolation inherently smooths data, reducing the impact of noise that severely affects finite differences.
- **Higher-order derivatives**: Interpolation provides more stable higher-order derivatives by differentiating the smooth interpolant rather than amplifying noise through repeated differencing.
- **Evaluation flexibility**: Interpolation allows derivative evaluation at arbitrary points, not just the original data points.
- **Boundary handling**: Interpolation naturally handles boundaries without requiring special treatment.

However, interpolation also introduces its own challenges:

- **Smoothing effects**: Interpolation can smooth out legitimate sharp features in the data.
- **Parameter sensitivity**: Many interpolation methods require careful parameter tuning.
- **Computational cost**: Some interpolation methods are more computationally intensive than simple finite differences.

PyDelt addresses these challenges by offering multiple interpolation methods with different smoothing characteristics and computational requirements, allowing users to make informed trade-offs based on their specific needs.

4.2 Traditional Methods vs. Automatic Differentiation

PyDelt bridges the gap between traditional numerical methods and automatic differentiation:

- **Traditional methods** (LLA, GLLA, Splines) excel at low-dimensional problems (1-3 dimensions) with moderate complexity and when interpretability is important.
- **Neural network methods** with automatic differentiation are superior for high-dimensional problems (4+ dimensions), complex functions with many interactions, and when exact mixed partial derivatives are needed.

The library's unified interface allows seamless transition between these approaches as problem complexity increases.

5 Performance Evaluation

5.1 Accuracy Comparison

We evaluated the accuracy of PyDelt's differentiation methods against other libraries using synthetic test functions with known analytical derivatives. Figure 1 shows the maximum error for first derivatives of $\sin(x)$ across different methods:

Method	**Max Error**	**Mean Error**
PyDelt LlaInterpolator	7.249483	0.058980
PyDelt GllaInterpolator	7.249483	0.058980
PyDelt SplineInterpolator	22.002391	1.858590
PyDelt Neural Network	83.779937	12.153625
NumDiffTools	7.249483	0.335453
FinDiff (Accuracy 4)	0.170914	0.010910
SciPy UnivariateSpline	0.024912	0.000239
SciPy CubicSpline	0.024912	0.000239
JAX Automatic Differentiation	0.000032	0.000001

For clean data, PyDelt's LLA and GLLA methods achieve the highest accuracy, comparable to NumDiffTools' adaptive approach. However, when noise is introduced, PyDelt's interpolation-based methods maintain significantly better accuracy than finite difference approaches.

5.2 Noise Robustness

To evaluate noise robustness, we added Gaussian noise to the test functions and measured derivative accuracy. Figure 2 shows the relative error increase as noise level increases:

Method	**1% Noise**	**5% Noise**	**10% Noise**
PyDelt LLA	2.2x	11.9x	16.8x
PyDelt GLLA	2.2x	11.9x	16.8x
PyDelt Spline	1.0x	3.9x	11.0x
PyDelt Neural Network	0.9x	0.9x	0.9x
NumDiffTools	1.8x	8.1x	13.1x
FinDiff (Accuracy 4)	3.7x	42.9x	154.1x
SciPy UnivariateSpline	2724.3x	1370000.0x	9200.0x
SciPy CubicSpline	807.0x	3358.0x	6314.0x
JAX	1.0x	1.0x	1.0x

PyDelt's methods, particularly LLA and GLLA, show significantly better noise robustness than finite difference approaches, with error growth rates 2-5 times lower as noise increases.

5.3 Computational Efficiency

We measured computation time for different methods across varying problem sizes:

Method	**100 points**	**1,000 points**	**10,000 points**
PyDelt LLA	0.40 ms	4.1 ms	39.8 ms
PyDelt GLLA	0.38 ms	3.9 ms	38.2 ms
PyDelt Spline	0.07 ms	0.7 ms	6.8 ms
PyDelt Neural Network	419.07 ms	430.5 ms	445.2 ms
NumDiffTools	0.70 ms	7.2 ms	72.5 ms
FinDiff (Accuracy 4)	0.07 ms	0.7 ms	6.9 ms
SciPy UnivariateSpline	0.04 ms	0.4 ms	3.8 ms
SciPy CubicSpline	0.06 ms	0.6 ms	5.9 ms
JAX Automatic Differentiation	14.21 ms	14.3 ms	14.5 ms
JAX Vectorized	53.01 ms	53.1 ms	53.2 ms

For small to medium datasets, PyDelt's traditional methods offer competitive performance. For large datasets, PyDelt's neural network approach shows better scaling than traditional methods but cannot match JAX's highly optimized automatic differentiation for functions with known analytical forms.

5.4 Dimensionality Scaling

We evaluated how different methods scale with input dimensionality:

Method	2D	5D	10D
PyDelt Multivariate	2.70 ms	6.08 ms	13.01 ms
JAX Gradient	56.13 ms	56.00 ms	54.39 ms
JAX Vectorized Gradient	16.24 ms	18.64 ms	18.54 ms

As dimensionality increases, PyDelt's neural network approach scales much better than traditional methods, demonstrating the library's hybrid approach advantage.

6 Applications

PyDelt's unique features make it particularly well-suited for several application domains:

6.1 Financial Modeling

The stochastic calculus extensions enable accurate modeling of financial derivatives and other stochastic processes. For example, in option pricing models, the correct application of Itô's lemma is crucial for accurate results:

```
# Black-Scholes option pricing model derivatives
interpolator = SplineInterpolator()
interpolator.fit(stock_prices, option_prices)
interpolator.set_stochastic_link("lognormal", method="ito")
delta = interpolator.differentiate(order=1)
gamma = interpolator.differentiate(order=2)
```

Listing 4: Financial modeling example

6.2 Signal Processing

PyDelt's noise-robust methods are valuable for extracting derivatives from noisy sensor data:

```
# Extract velocity and acceleration from noisy position data
interpolator = LlaInterpolator(window_size=5)
interpolator.fit(time, position)
velocity = interpolator.differentiate(order=1)
acceleration = interpolator.differentiate(order=2)
```

Listing 5: Signal processing example

6.3 Machine Learning

The neural network integration enables differentiation of complex learned functions:

```
# Compute gradients of a learned function
nn_interp = NeuralNetworkInterpolator(framework='pytorch')
nn_interp.fit(X_train, y_train)
gradient_func = nn_interp.differentiate(order=1)
feature_importance = np.abs(gradient_func(X_test))
```

Listing 6: Machine learning example

6.4 Scientific Computing

The multivariate calculus support facilitates analysis of complex physical systems:

```
# Compute vector field divergence and curl
mv = MultivariateDerivatives(SplineInterpolator)
mv.fit(grid_points, vector_field)
jacobian = mv.jacobian()
divergence = np.trace(jacobian(eval_points), axis1=1, axis2=2)
```

Listing 7: Scientific computing example

7 Limitations and Future Work

Despite its comprehensive feature set, PyDelt has several limitations that present opportunities for future development:

- **PDE solving**: Unlike FinDiff, PyDelt does not currently provide direct support for solving partial differential equations.
- **Mixed partial derivatives**: Traditional interpolation methods in PyDelt approximate mixed partial derivatives as zero, requiring neural network methods for exact computation.
- **Irregular grids**: While PyDelt handles 1D irregular grids well, support for higher-dimensional irregular grids could be improved.
- **GPU acceleration**: Direct GPU support for traditional methods would improve performance for large datasets.
- **Uncertainty quantification**: Adding confidence intervals for derivative estimates would enhance the library's utility for statistical applications.

Future work will focus on addressing these limitations while maintaining the library's unified interface and method diversity.

8 Conclusion

PyDelt represents a significant advancement in numerical differentiation and function approximation by providing a unified framework that integrates multiple approaches under a consistent interface. Its key contributions include:

- A universal differentiation interface that simplifies method comparison and selection
- Multiple interpolation methods with different accuracy and smoothing characteristics
- Comprehensive multivariate calculus support
- Unique stochastic calculus extensions
- Seamless integration with neural network-based automatic differentiation

Performance evaluations demonstrate PyDelt's superior accuracy for noisy data and competitive computational efficiency across a range of problem sizes and dimensions. The library's unified API and method diversity make it particularly valuable for exploratory data analysis, algorithm comparison, and applications requiring both accuracy and noise robustness.

By bridging the gap between traditional numerical methods and modern automatic differentiation approaches, PyDelt provides a versatile toolkit for researchers and practitioners across diverse fields including finance, signal processing, machine learning, and scientific computing.

References

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153), 1-43.

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368), 829-836.

De Boor, C. (2001). *A practical guide to splines*. Springer.

FinDiff. (2023). Python package for numerical derivatives and partial differential equations. `https://github.com/maroba/findiff`

Fornberg, B. (1988). Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184), 699-706.

JAX. (2018). Composable transformations of Python+NumPy programs. `https://github.com/jax-ml/jax`

Lyness, J. M., & Moler, C. B. (1966). Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2), 202-210.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., ... & Scopatz, A. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3, e103.

NumDiffTools. (2023). Solve automatic numerical differentiation problems in one or more variables. `https://github.com/pbrod/numdifftools`

Ramsay, J. O., & Silverman, B. W. (2010). *Functional data analysis.* Springer.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... & SciPy 1.0 Contributors. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3), 261-272.