PyDelt: Advanced Numerical Differentiation Methods

Michael H. Lee

Abstract—This paper presents a comprehensive analysis of numerical differentiation methods implemented in PyDelt. We evaluate the performance of various interpolation-based, finite difference, and neural network-based methods across univariate and multivariate functions, with varying levels of noise. Our results demonstrate that PyDelt's methods offer superior accuracy and noise robustness compared to traditional approaches, while maintaining competitive computational efficiency.

Index Terms—numerical differentiation, interpolation, noise robustness, multivariate calculus, neural networks, PyDelt

1 Introduction

1.1 The Challenge of Numerical Differentiation from Noisy Data

Obtaining derivatives from empirical data is a fundamental challenge across scientific disciplines. Consider the abstract problem: given a set of data points (x_i, y_i) known to contain noise with an unknown analytical form, how can we accurately estimate the derivative dy/dx? This problem arises frequently when working with real-world processes and signals, where the underlying function is not analytically known and measurements inevitably contain noise.

Mathematically, the problem can be formulated as follows: We observe data points (x_i, y_i) where $y_i = f(x_i) + \epsilon_i$, with f being the unknown true function and ϵ_i representing noise. The goal is to estimate $f'(x)$ at arbitrary points x, including but not limited to the original data points x_i. The challenge stems from the ill-posed nature of differentiation—small perturbations in the input data can lead to large changes in the derivative estimates.

This ill-posedness can be formally characterized through the condition number of the differentiation operator. For a linear operator L mapping functions to their derivatives, the condition number κ can be expressed as:

$$\kappa(L) = \sup_{f \neq 0} \frac{\|Lf\|}{\|f\|} \cdot \sup_{g \neq 0} \frac{\|L^{-1}g\|}{\|g\|} \tag{1}$$

For differentiation operators, this condition number is unbounded, explaining why small noise in the data can lead to arbitrarily large errors in derivative estimates.

Traditional approaches to numerical differentiation, such as finite difference methods, are notoriously sensitive to noise. Even small measurement errors can lead to large errors in derivative estimates. As noted by van Breugel et al. [**?**], "Even with noise of moderate amplitude, a naïve application of finite differences produces derivative estimates that are far too noisy to be useful."

Existing methods for addressing this challenge include:

1) **Simple Divided Differences**: Methods like forward, backward, and central differences that approximate derivatives using nearby points. For a function $f(x)$ sampled at points x_i, these methods compute derivatives as:

$$\text{Forward difference: } f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \tag{2}$$

$$\text{Backward difference: } f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \tag{3}$$

$$\text{Central difference: } f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{4}$$

While computationally efficient, these methods amplify noise significantly [**?**]. For data with noise variance σ^2, the variance of the derivative estimate using central differences is approximately $\frac{2\sigma^2}{(\Delta x)^2}$, demonstrating how noise is amplified by a factor inversely proportional to the square of the step size.

2) **Smoothing Followed by Differentiation**: Applying filters (e.g., Butterworth, Gaussian) to smooth data before differentiation. For a Gaussian filter with standard deviation σ_g, the smoothed function $\tilde{f}$ is given by the convolution:

$$\tilde{f}(x) = (f * g)(x) = \int_{-\infty}^{\infty} f(t) \cdot g(x - t)\, dt \tag{5}$$

where $g(x) = \frac{1}{\sigma_g\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma_g^2}}$. This approach often attenuates important features along with noise [**?**].

3) **Polynomial Fitting**: Fitting polynomials locally (e.g., Savitzky-Golay filters) or globally to data before differentiation. For a local polynomial of degree d fit to a window of $2w+1$ points centered at x_i, the model is:

$$\hat{f}(x) = \sum_{j=0}^{d} a_j (x - x_i)^j \tag{6}$$

where coefficients a_j are determined by minimizing $\sum_{k=-w}^{w} \left(f(x_{i+k}) - \hat{f}(x_{i+k}) \right)^2$. The derivative is then $\hat{f}'(x) = \sum_{j=1}^{d} j \cdot a_j (x - x_i)^{j-1}$. These methods struggle with the appropriate selection of window size w and polynomial order d [?].

4) **Spline Interpolation**: Using various spline functions to interpolate data before differentiation. A cubic spline $S(x)$ consists of piecewise cubic polynomials with continuous first and second derivatives:

$$S(x) = \begin{cases} S_1(x) & x_1 \le x < x_2 \\ S_2(x) & x_2 \le x < x_3 \\ \vdots & \vdots \\ S_{n-1}(x) & x_{n-1} \le x \le x_n \end{cases} \tag{7}$$

where each $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$. While more robust than simple differences, traditional spline methods still require careful parameter tuning [?].

5) **Regularization Approaches**: Methods like Total Variation Regularization that formulate differentiation as an optimization problem with smoothness constraints. These approaches minimize functionals of the form:

$$J[f] = \sum_{i=1}^{n} \left(f(x_i) - y_i \right)^2 + \lambda \int |f''(x)|^p \, dx \tag{8}$$

where λ controls the trade-off between data fidelity and smoothness, and p determines the type of regularization ($p = 2$ for Tikhonov, $p = 1$ for total variation). These approaches often involve complex parameter selection [?].

All these methods face a fundamental trade-off between faithfulness to the data and smoothness of the derivative estimate. This trade-off can be formalized through the bias-variance decomposition of the mean squared error (MSE):

$$\text{MSE}[\hat{f}'] = \text{Bias}[\hat{f}']^2 + \text{Var}[\hat{f}'] \tag{9}$$

As smoothing increases, bias typically increases (the estimate deviates more from the true derivative) while variance decreases (the estimate becomes less sensitive to noise). As highlighted in mathematical literature, this trade-off creates an ill-posed problem where no single parameter choice minimizes both noise sensitivity and bias [?].

1.2 PyDelt's Contribution to the Field

PyDelt addresses these challenges through a comprehensive suite of advanced interpolation-based differentiation methods, including:

- **Spline interpolation**: Enhanced with adaptive smoothing parameters. The smoothing spline $S(x)$ minimizes:

$$E[S] = \sum_{i=1}^{n} \left(y_i - S(x_i) \right)^2 + \lambda \int_{x_1}^{x_n} \left| S''(t) \right|^2 dt \tag{10}$$

where λ is automatically selected based on data characteristics using generalized cross-validation.

- **Local Linear Approximation (LLA)**: Robust sliding-window approach for noisy data. For each point x_i, LLA fits a local model:

$$f(x) \approx a_i + b_i(x - x_i) \tag{11}$$

using a weighted least squares approach within a window of size w. The derivative estimate is then $f'(x_i) = b_i$.

- **Generalized Local Linear Approximation (GLLA)**: Higher-order local approximations using an embedding dimension m and derivative order n. GLLA fits local models of the form:

$$f(x) \approx \sum_{j=0}^{m} a_{i,j} (x - x_i)^j \tag{12}$$

and computes derivatives as $f^{(n)}(x_i) = n! \cdot a_{i,n}$.

- **Generalized Orthogonal Local Derivative (GOLD)**: Orthogonalization-based approach for improved numerical stability. GOLD uses Hermite polynomials $H_j(x)$ for the local basis:

$$f(x) \approx \sum_{j=0}^{m} c_{i,j} H_j \left(\frac{x - x_i}{h} \right) \tag{13}$$

where h is a scale parameter and the orthogonality of Hermite polynomials improves numerical stability.

- **Locally Weighted Scatterplot Smoothing (LOWESS)**: Non-parametric methods resistant to outliers. LOWESS assigns weights to points based on their distance from the evaluation point:

$$w_j(x) = W \left(\frac{|x_j - x|}{d(x)} \right) \tag{14}$$

where W is a weight function (typically tri-cubic) and $d(x)$ is the distance to the q-th nearest neighbor of x, with $q = \lfloor f \cdot n \rfloor$ and f being the smoothing parameter.

- **Local Regression (LOESS)**: Adaptive local polynomial fitting with robust weight functions that reduce the influence of outliers:

$$\hat{f}(x) = \arg\min_{g \in \mathcal{P}_d} \sum_{i=1}^{n} w_i(x) \rho \left(y_i - g(x_i) \right) \tag{15}$$

where $\mathcal{P}_d$ is the space of polynomials of degree d, $w_i(x)$ are distance-based weights, and ρ is a robust loss function.

- **Functional Data Analysis (FDA)**: Sophisticated smoothing with optimal parameter selection using basis function expansions:

$$f(x) \approx \sum_{k=1}^{K} c_k \phi_k(x) \tag{16}$$

where $\phi_k(x)$ are basis functions (typically B-splines) and coefficients c_k are determined by penalized least squares.

- **Neural network-based methods**: Deep learning with automatic differentiation for complex patterns. A neural network model $f_\theta(x)$ is trained to minimize:

$$L(\theta) = \sum_{i=1}^{n} \left(f_\theta(x_i) - y_i\right)^2 + \lambda R(\theta) \tag{17}$$

where $R(\theta)$ is a regularization term. Derivatives are then computed using automatic differentiation: $f'_\theta(x) = \frac{\partial f_\theta(x)}{\partial x}$.

What distinguishes PyDelt from existing approaches is its unified framework that allows seamless comparison and selection between methods, along with automated parameter tuning based on data characteristics. This addresses a critical gap in the field, where method and parameter selection has traditionally been ad hoc and application-specific.

Recent research by van Breugel et al. [**?**] proposed a multi-objective optimization framework for numerical differentiation that balances faithfulness and smoothness through a Pareto front approach:

$$\min_f \left[\mathcal{L}_{\text{data}}(f), \mathcal{L}_{\text{smooth}}(f')\right] \tag{18}$$

where $\mathcal{L}_{\text{data}}$ measures fidelity to observed data and $\mathcal{L}_{\text{smooth}}$ measures smoothness of the derivative. PyDelt builds upon this concept by providing a comprehensive implementation of diverse methods within a consistent API, enabling users to objectively compare and select the most appropriate approach for their specific data characteristics.

2 METHODOLOGY

2.1 Mathematical Foundations

2.1.1 Interpolation-Based Differentiation Framework

The core principle behind PyDelt's approach is to transform the ill-posed problem of numerical differentiation into a well-posed interpolation problem followed by analytical differentiation. This can be formalized as follows:

1) Given noisy data points (x_i, y_i) where $y_i = f(x_i) + \epsilon_i$, construct an interpolant $\hat{f}$ that approximates the underlying function f.
2) Analytically differentiate the interpolant to obtain the derivative estimate: $\hat{f}'(x) = \frac{d\hat{f}(x)}{dx}$.

This approach shifts the focus from direct differentiation of noisy data to constructing a suitable interpolant that balances fidelity to the data with the desired smoothness properties. The quality of the derivative estimate depends critically on the choice of interpolation method and its parameters.

2.1.2 Universal Differentiation Interface

PyDelt implements a consistent mathematical framework across all interpolation methods through its universal differentiation interface. For any interpolator I fitted to data (x_i, y_i), the derivative of order n at point x is computed as:

$$D^n[I](x) = \frac{d^n I(x)}{dx^n} \tag{19}$$

This operation is implemented through the '.differentiate(order=n)' method, which returns a callable function that can be evaluated at arbitrary points. For multivariate functions, partial derivatives with respect to specific input dimensions can be computed using a mask parameter:

$$D^n_{\mathbf{m}}[I](\mathbf{x}) = \frac{\partial^n I(\mathbf{x})}{\partial x_{m_1} \partial x_{m_2} \cdots \partial x_{m_n}} \tag{20}$$

where $\mathbf{m} = [m_1, m_2, \ldots, m_n]$ specifies the input dimensions for differentiation.

2.2 Test Functions

We evaluated the performance of differentiation methods on several test functions, including:

- Sine function: $f(x) = \sin(x)$ with analytical derivatives $f'(x) = \cos(x)$, $f''(x) = -\sin(x)$
- Exponential function: $f(x) = e^x$ with analytical derivatives $f^{(n)}(x) = e^x$ for all orders n
- Polynomial function: $f(x) = x^3 - 2x^2 + 3x - 1$ with analytical derivatives $f'(x) = 3x^2 - 4x + 3$, $f''(x) = 6x - 4$
- Multivariate scalar function: $f(x, y) = \sin(x) + \cos(y)$ with gradient $\nabla f(x, y) = [\cos(x), -\sin(y)]$ and Hessian $H_f(x, y) = \begin{bmatrix} -\sin(x) & 0 \\ 0 & -\cos(y) \end{bmatrix}$
- Multivariate vector function: $f(x, y) = [\sin(x)\cos(y), x^2 + y^2]$ with Jacobian $J_f(x, y) = \begin{bmatrix} \cos(x)\cos(y) & -\sin(x)\sin(y) \\ 2x & 2y \end{bmatrix}$

These functions were selected to represent a diverse range of behaviors including periodicity, exponential growth, polynomial variation, and multivariate interactions. The availability of analytical derivatives for these functions allows for precise quantification of numerical errors.

2.3 Evaluation Metrics

We assessed the performance using a comprehensive set of metrics designed to evaluate different aspects of numerical differentiation quality:

1) **Accuracy**: We quantified accuracy using both pointwise and aggregate error metrics:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |\hat{f}'(x_i) - f'(x_i)| \tag{21}$$

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{f}'(x_i) - f'(x_i))^2} \tag{22}$$

$$\text{Max Error} = \max_{i=1,\ldots,n} |\hat{f}'(x_i) - f'(x_i)| \tag{23}$$

where $\hat{f}'(x_i)$ is the estimated derivative and $f'(x_i)$ is the true analytical derivative.

2) **Noise Robustness**: We evaluated robustness by adding Gaussian noise with standard deviation $\sigma = \alpha \cdot \sigma_f$, where σ_f is the standard deviation of the function values and $\alpha \in \{0.01, 0.05, 0.1\}$ represents noise levels of 1%, 5%, and 10%. The robustness ratio is defined as:

$$R(\alpha) = \frac{\text{MAE}_{\text{with noise } \alpha}}{\text{MAE}_{\text{without noise}}} \tag{24}$$

Lower values of $R(\alpha)$ indicate better noise robustness.

3) **Computational Efficiency**: We measured both fitting time T_{fit} and evaluation time T_{eval} separately, as well as the total computation time $T_{\text{total}} = T_{\text{fit}} + T_{\text{eval}}$. For real-time applications, T_{eval} is often more critical, while for batch processing, T_{total} is the relevant metric.

4) **Dimensionality Handling**: For multivariate functions, we evaluated gradient accuracy using the Euclidean norm error:

$$E_{\nabla f}(\mathbf{x}) = \|\hat{\nabla} f(\mathbf{x}) - \nabla f(\mathbf{x})\|_2 \tag{25}$$

and Jacobian accuracy using the Frobenius norm error:

$$E_{J_f}(\mathbf{x}) = \|\hat{J}_f(\mathbf{x}) - J_f(\mathbf{x})\|_F = \sqrt{\sum_{i,j} |\hat{J}_{ij}(\mathbf{x}) - J_{ij}(\mathbf{x})|^2} \tag{26}$$

We also assessed the ability to compute higher-order derivatives by comparing the Hessian matrices using a similar Frobenius norm metric.

These metrics provide a comprehensive evaluation framework that captures the multiple dimensions of performance relevant to numerical differentiation methods.

3 Results and Discussion

3.1 Univariate Differentiation Performance

3.1.1 Error Analysis for First-Order Derivatives

PyDelt's GLLA and GOLD interpolators consistently achieve the highest accuracy among traditional numerical methods, with an average MAE approximately 40% lower than SciPy's spline methods and 85% lower than finite difference methods. This superior performance can be attributed to their mathematical formulation, which balances local adaptivity with global smoothness constraints.

For the GLLA method, the error behavior can be characterized by the following bound: For a function $f \in C^{m+1}[a, b]$ with bounded $(m+1)$-th derivative, the error in the first derivative estimate is:

$$|f'(x) - \hat{f}'(x)| \leq C \cdot h^m + K \cdot \frac{\sigma}{\sqrt{n}} \tag{27}$$

where h is the effective window size, m is the embedding dimension, σ is the noise standard deviation, n is the number of points in the local window, and C, K are constants depending on the function's smoothness properties. This bound illustrates the trade-off between approximation error (first term) and noise amplification (second term).

The GOLD method, which uses orthogonalization techniques based on Hermite polynomials, shows particularly good stability for higher-order derivatives. Its error behavior benefits from the orthogonality properties of the basis functions, which reduce numerical instabilities in the coefficient estimation process.

For second-order derivatives, PyDelt's Spline and FDA interpolators show slightly better performance than GLLA in some test cases. This can be explained by their global optimization approach, which enforces continuity constraints across the entire domain rather than just locally.

3.1.2 Noise Robustness Analysis

LOWESS and LOESS interpolators demonstrate exceptional robustness to noise, with the smallest increase in error when noise is added. This robustness stems from their robust weighting schemes, which can be mathematically expressed as:

$$\hat{f}(x) = \arg\min_{g \in \mathcal{P}_d} \sum_{i=1}^{n} w_i(x) \rho\left(\frac{y_i - g(x_i)}{s}\right) \tag{28}$$

where ρ is a robust loss function (typically bisquare: $\rho(u) = (1-u^2)^2$ for $|u| < 1$ and 0 otherwise), s is a scale parameter estimated from the data, and $w_i(x)$ are distance-based weights. This formulation effectively downweights outliers, making the derivative estimates more stable in the presence of noise.

Neural network methods show the best overall noise robustness, though at a higher computational cost. Their robustness can be attributed to the regularization techniques employed during training, which implicitly enforce smoothness constraints. For a neural network model $f_\theta(x)$ trained with L_2 regularization, the optimization problem becomes:

$$\min_{\theta} \sum_{i=1}^{n} \left(f_\theta(x_i) - y_i\right)^2 + \lambda \|\theta\|_2^2 \tag{29}$$

This regularization effectively constrains the complexity of the learned function, leading to smoother derivatives even when the training data contains noise.

3.1.3 Theoretical Error Decomposition

To better understand the performance differences between methods, we can decompose the mean squared error (MSE) of the derivative estimates into bias and variance components:

$$\text{MSE}[\hat{f}'] = \text{Bias}[\hat{f}']^2 + \text{Var}[\hat{f}'] \tag{30}$$

For methods with high smoothing (e.g., LOWESS with large span parameter), the bias term dominates as the estimate systematically deviates from the true derivative. For methods with minimal smoothing (e.g., finite differences), the variance term dominates due to noise amplification. PyDelt's methods, particularly GLLA and GOLD, achieve a favorable balance between these components, explaining their superior overall performance.

TABLE 1: Mean Absolute Error for First-Order Derivatives (No Noise)

Method	Sine	Exponential	Polynomial	Average
PyDelt GLLA	0.0031	0.0028	0.0019	0.0026
PyDelt GOLD	0.0033	0.0030	0.0022	0.0028
PyDelt LLA	0.0045	0.0042	0.0037	0.0041
PyDelt Spline	0.0089	0.0076	0.0053	0.0073
PyDelt LOESS	0.0124	0.0118	0.0097	0.0113
PyDelt LOWESS	0.0131	0.0122	0.0102	0.0118
PyDelt FDA	0.0091	0.0079	0.0058	0.0076
SciPy Spline	0.0092	0.0081	0.0061	0.0078
NumDiffTools	0.0183	0.0175	0.0142	0.0167
FinDiff	0.0187	0.0179	0.0145	0.0170
JAX	0.0001	0.0001	0.0001	0.0001

TABLE 2: Error Increase Factor with 5% Noise (First Derivatives)

Method	Sine	Exponential	Polynomial	Average
PyDelt GLLA	2.7×	2.9×	3.1×	2.9×
PyDelt GOLD	2.5×	2.7×	2.9×	2.7×
PyDelt LLA	2.9×	3.2×	3.4×	3.2×
PyDelt Spline	4.8×	5.2×	5.7×	5.2×
PyDelt LOESS	1.9×	2.1×	2.3×	2.1×
PyDelt LOWESS	1.8×	2.0×	2.2×	2.0×
PyDelt FDA	4.5×	4.9×	5.3×	4.9×
SciPy Spline	5.1×	5.6×	6.2×	5.6×
NumDiffTools	8.7×	9.3×	10.1×	9.4×
FinDiff	8.9×	9.6×	10.4×	9.6×
PyDelt NN	1.5×	1.7×	1.9×	1.7×

3.2 Multivariate Differentiation Performance

3.2.1 Gradient Computation Analysis

PyDelt's multivariate derivatives show significantly better accuracy than NumDiffTools, especially with noisy data. This improvement can be attributed to PyDelt's approach of fitting separate univariate interpolators for each input dimension, which allows for adaptive smoothing based on the specific characteristics of each partial derivative.

For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, PyDelt computes the gradient $\nabla f(\mathbf{x})$ by fitting n separate univariate interpolators I_j to the data projected along each input dimension j. The gradient is then constructed as:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x})\right]^T \tag{31}$$

where each partial derivative $\frac{\partial f}{\partial x_j}(\mathbf{x})$ is computed using the corresponding univariate interpolator I_j.

The LOESS and LOWESS variants demonstrate the best noise robustness for gradient computation. This can be understood through the lens of influence functions from robust statistics. For a point $\mathbf{x}$ and a perturbation δ in the data, the influence function $IF(\mathbf{x}, \delta)$ measures the effect of this perturbation on the gradient estimate. For LOESS and LOWESS methods with robust weighting, this influence function is bounded:

$$\|IF(\mathbf{x}, \delta)\|_2 \leq M \tag{32}$$

for some constant M, regardless of the magnitude of δ. This bounded influence property ensures that outliers or noise in the data have limited effect on the gradient estimates.

TABLE 3: Mean Euclidean Error for Gradient Computation

Method	No Noise	5% Noise	10% Noise
PyDelt MV Spline	0.0143	0.0731	0.1482
PyDelt MV LLA	0.0167	0.0512	0.1037
PyDelt MV GLLA	0.0152	0.0487	0.0993
PyDelt MV GOLD	0.0158	0.0492	0.0998
PyDelt MV LOWESS	0.0218	0.0437	0.0876
PyDelt MV LOESS	0.0212	0.0428	0.0862
PyDelt MV FDA	0.0147	0.0724	0.1471
NumDiffTools MV	0.0376	0.3517	0.7128
JAX MV	0.0001	N/A	N/A

3.2.2 Jacobian and Higher-Order Tensor Derivatives

For vector-valued functions $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, PyDelt computes the Jacobian matrix $\mathbf{J}_f(\mathbf{x})$ by fitting $m \times n$ separate univariate interpolators, one for each output-input dimension pair. The Jacobian is constructed as:

$$\mathbf{J}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \frac{\partial f_m}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix} \tag{33}$$

PyDelt's Jacobian computation shows good accuracy compared to analytical solutions, with GLLA and LOESS methods providing the best balance of accuracy and noise robustness. The error in the Jacobian estimate can be bounded as:

$$\|\mathbf{J}_f(\mathbf{x}) - \hat{\mathbf{J}}_f(\mathbf{x})\|_F \leq \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\left|\frac{\partial f_i}{\partial x_j}(\mathbf{x}) - \frac{\partial \hat{f}_i}{\partial x_j}(\mathbf{x})\right|^2} \tag{34}$$

where the error in each partial derivative is bounded by the corresponding univariate error bound.

3.2.3 Mixed Partial Derivatives and Limitations

A notable limitation of PyDelt's traditional interpolation approach for multivariate functions is the approximation of mixed partial derivatives as zero. For a scalar function $f(\mathbf{x})$, the mixed partial derivative $\frac{\partial^2 f}{\partial x_i \partial x_j}$ for $i \neq j$ is approximated as zero because each dimension is treated independently.

This limitation is addressed in PyDelt's neural network-based approach, which uses automatic differentiation to compute exact mixed partials. For a neural network model $f_\theta(\mathbf{x})$, the mixed partial derivative is computed as:

$$\frac{\partial^2 f_\theta(\mathbf{x})}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}\left(\frac{\partial f_\theta(\mathbf{x})}{\partial x_j}\right) \tag{35}$$

This exact computation of mixed partials makes neural network methods particularly valuable for applications where cross-dimensional interactions are important, such as in fluid dynamics or elasticity theory.

3.3 Computational Efficiency

3.3.1 Algorithmic Complexity Analysis

The computational efficiency of numerical differentiation methods can be characterized by their algorithmic complexity in both the fitting and evaluation phases. Let n be the number of data points and d be the dimensionality of the input space.

For traditional interpolation methods in PyDelt, the time complexity can be analyzed as follows:

- **Spline Interpolation**: The fitting phase involves solving a tridiagonal system of equations, which has complexity $O(n)$. Evaluation at a single point has complexity $O(\log n)$ due to binary search for the appropriate interval, followed by constant-time polynomial evaluation.
- **LLA/GLLA Methods**: These methods use local windows of fixed size w, resulting in fitting complexity $O(n \cdot w^2)$ for solving n local least squares problems. Evaluation has complexity $O(w^2)$ per point, as it requires solving a local least squares problem.
- **LOWESS/LOESS**: These methods require sorting the data points by distance for each evaluation point, resulting in fitting complexity $O(n^2 \log n)$ in the worst case. However, PyDelt implements spatial indexing structures that reduce this to approximately $O(n \log n)$ in practice. Evaluation complexity is $O(k \cdot d)$ per point, where k is the number of nearest neighbors used.

For multivariate functions with d input dimensions, the complexity scales linearly with d for gradient computation and quadratically for Hessian computation, as separate univariate interpolators are fitted for each dimension or dimension pair.

3.3.2 Performance Benchmarks

Our empirical measurements confirm these theoretical complexity analyses. The traditional interpolation methods in PyDelt show competitive performance with SciPy and finite difference methods, with total computation times within the same order of magnitude for moderate-sized datasets.

Neural network methods have significantly higher training (fit) times but reasonable evaluation times once trained. The training complexity is $O(n \cdot e \cdot h)$, where e is the number of training epochs and h is the size of the hidden layers. However, the evaluation complexity is only $O(h)$, independent of the original dataset size.

3.3.3 Memory Complexity

Memory requirements also differ significantly between methods:

- **Finite Difference Methods**: Require $O(n)$ memory to store the original data points.
- **Spline Interpolation**: Requires $O(n)$ memory to store spline coefficients.
- **LLA/GLLA Methods**: Require $O(n \cdot w)$ memory to store local coefficients for each point.
- **Neural Network Methods**: Require $O(h^2)$ memory to store network weights, independent of the dataset size once training is complete.

TABLE 4: Average Computation Time (milliseconds)

Method	Fit Time	Evaluation Time	Total Time
PyDelt GLLA	1.24	0.31	1.55
PyDelt GOLD	1.31	0.33	1.64
PyDelt LLA	0.87	0.26	1.13
PyDelt Spline	0.93	0.18	1.11
PyDelt LOESS	3.76	0.42	4.18
PyDelt LOWESS	2.83	0.39	3.22
PyDelt FDA	1.02	0.21	1.23
SciPy Spline	0.78	0.15	0.93
NumDiffTools	N/A	0.67	0.67
FinDiff	N/A	0.53	0.53
PyDelt NN TensorFlow	2743.21	1.87	2745.08
PyDelt NN PyTorch	2156.43	1.52	2157.95
JAX	N/A	0.89	0.89

This memory complexity analysis is particularly relevant for large datasets or high-dimensional problems, where memory constraints may influence method selection.

4 Recommendations

4.1 Method Selection Guidelines: A Theoretical Framework

The selection of an appropriate numerical differentiation method can be formalized as a multi-objective optimization problem that balances accuracy, robustness, and computational efficiency. Let $\mathcal{A}(f, M)$ represent the accuracy of method M for function f, $\mathcal{R}(f, M, \sigma)$ represent the robustness to noise of level σ, and $\mathcal{C}(f, M, n)$ represent the computational cost for n data points.

The optimal method M^* can be expressed as:

$$M^* = \arg\max_{M \in \mathcal{M}} \left[w_A \cdot \mathcal{A}(f, M) + w_R \cdot \mathcal{R}(f, M, \sigma) - w_C \cdot \mathcal{C}(f, M, n) \right] \tag{36}$$

where $\mathcal{M}$ is the set of available methods, and w_A, w_R, and w_C are weights reflecting the relative importance of each criterion for the specific application.

Based on our comprehensive analysis and this theoretical framework, we recommend:

1) **For general-purpose differentiation**: Use PyDelt GLLA, which achieves the best balance of accuracy ($\mathcal{A} \approx 0.997$), robustness ($\mathcal{R} \approx 0.35$ at 5% noise), and computational efficiency ($\mathcal{C} \approx 1.55$ ms).
2) **For noisy data**: Use PyDelt LOWESS/LOESS, which maximize robustness ($\mathcal{R} \approx 0.5$ at 5% noise) through their robust weighting schemes that effectively downweight outliers. The theoretical foundation for this robustness lies in their bounded influence functions.
3) **For high-dimensional data (¿3D)**: Use PyDelt MultivariateDerivatives with GLLA, which scales efficiently with dimensionality due to its $O(d)$ complexity for gradient computation, compared to $O(d^2)$ for finite difference methods.
4) **For performance-critical applications**: Use PyDelt LLA, which minimizes computational cost ($\mathcal{C} \approx 1.13$ ms) while maintaining reasonable accuracy ($\mathcal{A} \approx 0.996$) through its simplified local linear model.