PyDelt: Advanced Numerical Function Approximation, Differentiation, & Integration

Michael H. Lee

September 23, 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 across scientific computing, financial modeling, engineering, and data science domains.

Contents

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 notoriously 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.

In this paper, we present a comprehensive analysis of PyDelt's numerical differentiation methods compared to other popular libraries. 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.

2 Advanced Calculus Implementations

2.1 Multivariate Calculus Implementation

PyDelt's multivariate calculus module provides a unified framework for computing derivatives of functions with multiple input and output variables. The implementation follows the same design principles as the univariate module, with a consistent API and support for multiple interpolation methods.

2.1.1 Design Approach

The multivariate derivatives module is built around the `MultivariateDerivatives` class, which takes any PyDelt interpolator as a base estimator. The implementation strategy involves:

1. **Dimension Separation**: For each output dimension and input dimension pair, a separate univariate interpolator is fitted.
2. **Universal Interface**: The class provides methods for computing gradients, Jacobians, Hessians, and Laplacians, all following a consistent pattern of returning callable functions.
3. **Automatic Reshaping**: Input and output data are automatically reshaped to handle both 1D and 2D arrays, simplifying the API for users.

4. **Efficient Evaluation**: Derivatives are computed only when needed, with caching of intermediate results for performance.

The implementation supports both scalar-valued functions (returning gradients and Hessians) and vector-valued functions (returning Jacobians), making it versatile for a wide range of applications.

2.1.2 Mathematical Formulation

For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is computed as:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right]^T \tag{1}$$

where each partial derivative $\frac{\partial f}{\partial x_i}$ is computed using a separate univariate interpolator.

For a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian matrix is computed as:

$$\mathbf{J}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \frac{\partial f_m}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix} \tag{2}$$

where each element $\frac{\partial f_i}{\partial x_j}$ is computed using a separate univariate interpolator.

2.1.3 Limitations and Considerations

The primary limitation of this approach is that mixed partial derivatives (e.g., $\frac{\partial^2 f}{\partial x_i \partial x_j}$ for $i \neq j$) are approximated as zero for traditional interpolation methods. This is because each dimension is treated independently, and the interpolators cannot capture cross-dimensional interactions directly.

For applications requiring accurate mixed partial derivatives, PyDelt offers neural network-based methods that use automatic differentiation to compute exact mixed partials. These methods fit a neural network to the data and then use automatic differentiation to compute derivatives of any order.

2.2 Tensor Calculus Operations

PyDelt extends beyond basic multivariate calculus to support tensor calculus operations, which are essential for applications in continuum mechanics, fluid dynamics, and other fields involving tensor fields.

2.2.1 Tensor Field Operations

The tensor calculus module supports the following operations:

1. **Vector Field Gradient**: For a vector field $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$, computes the tensor field $\nabla\mathbf{F}$ with components $(\nabla\mathbf{F})_{ijk} = \frac{\partial F_i}{\partial x_j}$.

2. **Divergence**: For a vector field $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$, computes the scalar field $\nabla \cdot \mathbf{F} = \sum_{i=1}^{n} \frac{\partial F_i}{\partial x_i}$.

3. **Curl**: For a vector field $\mathbf{F} : \mathbb{R}^3 \to \mathbb{R}^3$, computes the vector field $\nabla \times \mathbf{F}$ with components $(\nabla \times \mathbf{F})_i = \epsilon_{ijk} \frac{\partial F_k}{\partial x_j}$, where ϵ_{ijk} is the Levi-Civita symbol.

4. **Tensor Contraction**: For a tensor field $\mathbf{T} : \mathbb{R}^n \to \mathbb{R}^{m \times p}$, computes contractions such as the trace $\text{tr}(\mathbf{T}) = \sum_{i=1}^{\min(m,p)} T_{ii}$.

2.2.2 Implementation Approach

Tensor calculus operations are implemented using the multivariate derivatives module as a foundation, with additional functions for tensor-specific operations. The implementation uses NumPy's array operations for efficient computation and maintains proper tensor dimensions throughout.

For applications requiring coordinate transformations, PyDelt provides utilities for converting between different coordinate systems (e.g., Cartesian, spherical, cylindrical) and transforming tensor components accordingly.

2.3 Stochastic Calculus

PyDelt's stochastic calculus module addresses the unique challenges of computing derivatives for stochastic processes, where traditional calculus rules do not apply due to the non-differentiable nature of sample paths.

2.3.1 Itô Calculus

Itô calculus is a framework for computing derivatives of functions of stochastic processes. The key insight is that for a function f of a stochastic process X_t, the differential $df(X_t)$ includes an additional term due to the quadratic variation of the process:

$$df(X_t) = f'(X_t)\, dX_t + \frac{1}{2} f''(X_t)\, (dX_t)^2 \tag{3}$$

where $(dX_t)^2$ is evaluated using the quadratic variation of the process (e.g., dt for standard Brownian motion).

PyDelt implements Itô corrections by extending the standard interpolators with a `set_stochastic_link` method that specifies the distribution

and calculus type. The implementation then automatically applies the appropriate correction terms when computing derivatives.

2.3.2 Stratonovich Calculus

Stratonovich calculus provides an alternative interpretation of stochastic integrals that preserves the ordinary chain rule of calculus. For a function f of a stochastic process X_t, the Stratonovich differential is:

$$df(X_t) = f'(X_t) \circ dX_t \tag{4}$$

where $\circ$ denotes the Stratonovich integral.

PyDelt supports both Itô and Stratonovich calculus, allowing users to choose the most appropriate framework for their application. The implementation handles the conversion between the two frameworks automatically.

2.3.3 Stochastic Link Functions

PyDelt's stochastic calculus module supports various link functions that transform derivatives through probability distributions:

1. **Normal**: For normally distributed processes with constant volatility.
2. **Log-normal**: For processes where the logarithm follows a normal distribution, commonly used in financial modeling.
3. **Gamma**: For processes with gamma-distributed increments.
4. **Beta**: For processes constrained to a finite interval.
5. **Exponential**: For processes with exponentially distributed waiting times.
6. **Poisson**: For counting processes with independent increments.

Each link function implements the appropriate correction terms for both Itô and Stratonovich calculus, ensuring accurate derivatives for the corresponding stochastic process.

2.4 Integration with Neural Networks

PyDelt integrates neural networks with automatic differentiation to provide exact derivatives for complex functions and overcome the limitations of traditional interpolation methods.

2.4.1 Neural Network Interpolation

The `NeuralNetworkInterpolator` class fits a neural network to the input-output data and uses automatic differentiation to compute derivatives of any order. The implementation supports both PyTorch and TensorFlow backends, allowing users to leverage their preferred deep learning framework.

The neural network approach offers several advantages:

1. **Exact Mixed Partials**: Automatic differentiation computes exact mixed partial derivatives, overcoming the primary limitation of traditional interpolation methods.
2. **Complex Relationships**: Neural networks can capture complex, non-linear relationships that may be difficult to model with traditional interpolation methods.
3. **GPU Acceleration**: Neural network computations can be accelerated using GPUs, providing significant performance improvements for large datasets.

The trade-off is increased computational cost during training and the need for careful hyperparameter tuning to avoid overfitting or underfitting.

2.4.2 Physics-Informed Neural Networks

PyDelt also supports physics-informed neural networks (PINNs), which incorporate physical laws expressed as differential equations into the neural network training process. This approach allows for solving both forward and inverse problems involving partial differential equations.

PINNs are particularly valuable for applications where the underlying physical laws are known but the data may be sparse or noisy. By incorporating the differential equations as constraints during training, PINNs can produce solutions that are both data-consistent and physically plausible.

3 Motivation and Applications

The development of PyDelt was motivated by several key challenges in numerical differentiation that remain inadequately addressed by existing tools. These challenges span theoretical, practical, and software engineering concerns.

3.1 Key Motivations

3.1.1 Noise Sensitivity in Traditional Methods

Numerical differentiation is inherently an ill-posed problem—small perturbations in input data can lead to large changes in derivative estimates. This

sensitivity is particularly problematic when working with real-world data, which invariably contains measurement noise, sampling irregularities, and other imperfections. Traditional finite difference methods amplify these errors, often rendering derivative estimates unusable without extensive preprocessing [Fornberg, 1988].

PyDelt addresses this challenge through its focus on interpolation-based methods that inherently incorporate smoothing. Methods like LOWESS and LOESS are specifically designed to be robust against outliers and varying noise levels [Cleveland, 1979], while spline-based approaches with adjustable smoothing parameters allow users to explicitly control the trade-off between fidelity to data and smoothness of derivatives.

3.1.2 Method Selection Complexity

Choosing the appropriate differentiation method for a specific application requires considerable expertise. Different methods excel under different conditions: some perform better with noisy data, others with sparse sampling, and still others with specific types of underlying functions. This complexity creates a significant barrier to entry for many potential users.

PyDelt's universal interface allows users to easily compare multiple methods on their specific data, facilitating empirical method selection without requiring deep theoretical understanding of each approach. The consistent '.fit().differentiate()' pattern means that switching between methods requires changing only a single line of code, encouraging experimentation and comparison.

3.1.3 Fragmentation of Implementations

Existing numerical differentiation tools are scattered across different libraries, each with its own API, assumptions, and limitations. This fragmentation makes it difficult to compare methods fairly and to integrate differentiation into larger workflows.

PyDelt unifies diverse approaches under a single, consistent interface, allowing users to focus on their application rather than on the idiosyncrasies of different implementations. This unification extends beyond traditional methods to include neural network-based approaches and stochastic calculus extensions, providing a comprehensive toolkit within a single framework.

3.1.4 Limited Support for Advanced Use Cases

Many existing tools focus on basic univariate differentiation, with limited support for multivariate calculus, stochastic processes, or integration with modern machine learning frameworks. This limitation forces users to implement custom solutions or to cobble together multiple tools for advanced applications.

PyDelt explicitly addresses these advanced use cases, providing built-in support for multivariate calculus operations (gradients, Jacobians, Hessians), stochastic calculus corrections, and seamless integration with deep learning frameworks like PyTorch and TensorFlow.

3.2 Application Domains

PyDelt's comprehensive approach to numerical differentiation makes it valuable across a wide range of application domains:

3.2.1 Scientific Computing

In scientific computing, PyDelt enables:

- **Differential Equation Discovery**: Extracting governing equations from experimental data by accurately estimating derivatives and fitting them to candidate equation forms [Raissi et al., 2019].
- **Phase Space Analysis**: Reconstructing phase spaces from time series data by computing derivatives to reveal underlying dynamics in nonlinear systems [Boker and Nesselroade, 2002].
- **Fluid Dynamics**: Computing properties like vorticity, strain rates, and divergence from velocity field measurements, which require accurate spatial derivatives [Shu, 1998].

3.2.2 Financial Modeling

In finance, PyDelt supports:

- **Option Greeks Calculation**: Computing sensitivity measures (Delta, Gamma, Theta, Vega) that are derivatives of option prices with respect to various parameters [Cont and Tankov, 2004].
- **Volatility Surface Modeling**: Estimating implied volatility and its derivatives across strike prices and maturities to understand market dynamics [Aït-Sahalia and Jacod, 2014].
- **Risk Management**: Applying proper stochastic calculus corrections when working with financial time series, which often follow stochastic processes rather than deterministic functions.

3.2.3 Engineering Design

Engineers can use PyDelt for:

- **System Identification**: Extracting dynamic models from sensor data by estimating derivatives and fitting them to differential equations [Solak et al., 2003].
- **Control Design**: Developing controllers that require derivative feedback, such as PID controllers or model predictive control systems.
- **Optimization**: Implementing gradient-based optimization methods that require accurate derivatives of objective functions and constraints.

3.2.4 Data Science

Data scientists benefit from PyDelt through:

- **Feature Engineering**: Creating derivative-based features that capture rates of change and higher-order dynamics in time series data.
- **Signal Processing**: Enhancing signal analysis with robust derivatives that are less sensitive to noise than traditional methods.
- **Model Validation**: Verifying that machine learning models satisfy known physical constraints by checking derivatives and their relationships.

3.2.5 Environmental Science

Environmental scientists can leverage PyDelt for:

- **Climate Data Analysis**: Detecting acceleration in temperature changes, sea level rise, or ice melt rates from long-term monitoring data [Piao et al., 2020].
- **Ecological Modeling**: Calculating growth rates, carrying capacities, and predator-prey interactions from population count time series [Clark et al., 2001].
- **Hydrological Studies**: Estimating infiltration rates, groundwater recharge, and river discharge dynamics from water level time series.

These diverse applications highlight the broad utility of PyDelt's approach to numerical differentiation, which combines theoretical rigor with practical usability across domains.

4 Related Work

Several software libraries provide numerical differentiation capabilities, each with different approaches and strengths. This section reviews the most prominent alternatives to PyDelt and highlights their relative advantages and limitations.

4.1 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 approach requires manual chaining
from scipy.interpolate import UnivariateSpline
spline = UnivariateSpline(x, y, s=0.1)
derivative = spline.derivative(n=1)
result = derivative(x_eval)
```

Listing 1: SciPy differentiation example

SciPy's approach focuses on interpolation quality rather than derivative accuracy, with limited guidance for method selection based on derivative requirements. Additionally, while SciPy provides excellent univariate interpolation, its support for multivariate derivatives is more limited and lacks a unified interface across different operations.

4.2 NumDiffTools

NumDiffTools [Brodtkorb, 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.

```
# NumDiffTools uses finite differences
import numdifftools as nd

# For univariate functions
df = nd.Derivative(lambda x: np.sin(x), n=1)
result = df(x_eval)

# For multivariate functions
gradient = nd.Gradient(lambda x: np.sin(x[0]) + np.cos(x[1]))
result = gradient([1.0, 2.0])
```

Listing 2: NumDiffTools example

However, NumDiffTools' 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. Its strength lies in computing accurate derivatives of analytical functions rather than noisy data.

4.3 FinDiff

FinDiff [Ritter, 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.

```
# FinDiff uses finite differences on grids
from findiff import FinDiff

# 1D derivative
d_dx = FinDiff(0, dx, 1)  # 1st derivative in x-direction
result = d_dx(y)

# 2D gradient
d_dx = FinDiff(0, dx, 1)  # d/dx
d_dy = FinDiff(1, dy, 1)  # d/dy
gradient = [d_dx(f), d_dy(f)]
```

Listing 3: FinDiff example

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. FinDiff is particularly well-suited for solving partial differential equations on structured grids but less ideal for general derivative estimation from noisy data.

4.4 JAX

JAX [Team, 2018] provides automatic differentiation capabilities through its `jax.grad` function, allowing exact computation of derivatives for functions defined in Python.

```
# JAX uses automatic differentiation
import jax
import jax.numpy as jnp

# Define function
def f(x):
    return jnp.sin(x)

# Get gradient function
df_dx = jax.grad(f)

# Evaluate gradient
result = df_dx(2.0)
```

Listing 4: JAX automatic differentiation example

JAX 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.

4.5 SymPy

SymPy [Meurer et al., 2017] offers symbolic differentiation through its `sympy.diff` function, computing exact derivatives of mathematical expressions.

```
# SymPy uses symbolic differentiation
from sympy import symbols, diff, sin

# Define symbolic variable and function
x = symbols('x')
f = sin(x)

# Compute derivative symbolically
df_dx = diff(f, x)

# Evaluate at specific point
result = df_dx.subs(x, 2.0).evalf()
```

Listing 5: SymPy symbolic differentiation example

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.

4.6 Specialized Time Series Derivative Methods

Several specialized methods have been developed for time series derivative estimation, particularly in fields like psychology, economics, and biomedicine:

- **Local Linear Approximation (LLA)** was introduced by Boker and Nesselroade [2002] for estimating derivatives from time series data, particularly for modeling oscillatory systems.
- **Generalized Local Linear Approximation (GLLA)** was developed by Boker et al. [2010] to extend LLA to higher-order derivatives using a generalized linear approximation framework.
- **Functional Data Analysis (FDA)** approaches, as described by Ramsay and Silverman [2005], use spline-based smoothing to represent time series as continuous functions from which derivatives can be analytically computed.

These methods have been implemented in various R packages (e.g., `doremi`, `fda`, `EGAnet`) but lacked a comprehensive Python implementation before PyDelt.

4.7 Neural Network Approaches

Recent research has explored using neural networks for derivative estimation:

- **Neural Ordinary Differential Equations** [Chen et al., 2018] use neural networks to learn the dynamics of continuous-time systems, parameterizing the derivative of the hidden state using a neural network.
- **Physics-Informed Neural Networks** [Raissi et al., 2019] incorporate physical laws expressed as differential equations into the learning process, allowing for solving both forward and inverse problems involving nonlinear partial differential equations.

These approaches leverage automatic differentiation to enforce differential constraints during training but typically require specialized implementations for each application.

4.8 Comparison to PyDelt

PyDelt distinguishes itself from these existing tools in several key ways:

- **Unified Interface**: Unlike SciPy, NumDiffTools, or FinDiff, PyDelt provides a consistent API across all methods, simplifying method comparison and selection.
- **Method Diversity**: PyDelt integrates traditional interpolation methods, specialized time series techniques (LLA, GLLA, FDA), and neural network approaches under a single framework.
- **Noise Robustness**: Unlike finite difference methods in NumDiffTools and FinDiff, PyDelt emphasizes interpolation-based approaches that are inherently more robust to noise.
- **Comprehensive Multivariate Support**: PyDelt provides full support for multivariate calculus operations (gradients, Jacobians, Hessians) with a consistent interface.
- **Stochastic Extensions**: Unique among numerical differentiation libraries, PyDelt includes stochastic calculus corrections for financial and other stochastic applications.
- **Neural Network Integration**: PyDelt seamlessly integrates with deep learning frameworks, bridging the gap between traditional numerical methods and modern automatic differentiation approaches.

These distinctions position PyDelt as a comprehensive solution for numerical differentiation across a wide range of applications, from basic univariate problems to complex multivariate and stochastic scenarios.

5 PyDelt: Design and Features

PyDelt was designed to address the limitations of existing tools by providing a comprehensive, unified framework for numerical differentiation and function approximation. The library's architecture is built around several core principles and features that distinguish it from other numerical differentiation tools.

5.1 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 6: Universal differentiation interface example

This interface design follows several key principles:

- **Separation of Concerns**: The `fit()` method handles data preprocessing and model fitting, while `differentiate()` focuses solely on derivative computation.
- **Lazy Evaluation**: The `differentiate()` method returns a callable function rather than immediately computing derivatives, allowing for efficient evaluation at arbitrary points.
- **Consistent Parameters**: All interpolators accept the same core parameters (`order` for derivative order, `mask` for partial derivatives), simplifying method switching.
- **Method Chaining**: The fluent interface supports method chaining (e.g., `interpolator.fit(x, y).differentiate(order=1)`), enhancing code readability.

This universal interface extends across all interpolation methods, from basic splines to sophisticated neural networks, providing a consistent experience regardless of the underlying algorithm.

5.2 Multiple Interpolation Methods

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

- **SplineInterpolator**: Creates smooth, continuous piecewise polynomial functions with controllable smoothing. Particularly effective for smooth underlying functions with moderate noise.
- **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 [Boker and Nesselroade, 2002].
- **Generalized Local Linear Approximation (GLLA)**: Extends LLA to higher-order derivatives using a generalized linear approximation framework [Boker et al., 2010].
- **GOLD (Generalized Optimal Linear Derivative)**: Implements Hermite cubic interpolation for smooth derivatives with optimal error properties. Particularly effective for data with high curvature regions.
- **Functional Data Analysis (FDA)**: Uses spline-based smoothing with automatic parameter selection based on data characteristics [Ramsay and Silverman, 2005].
- **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. Each method implements the same universal interface, making it easy to compare their performance on a given dataset.

5.3 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 7: Multivariate calculus example

The multivariate module maintains the same design philosophy as the core interpolation methods, with a consistent interface and lazy evaluation. It supports arbitrary evaluation points and handles both scalar and vector-valued functions.

5.4 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 8: Stochastic calculus example

These extensions are particularly valuable for financial applications, where standard calculus rules do not apply due to the non-differentiable nature of stochastic processes. By incorporating proper stochastic corrections, PyDelt ensures accurate derivative estimates for financial time series and other stochastic data.

5.5 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.

```
# Neural network interpolation with automatic differentiation
from pydelt.interpolation import NeuralNetworkInterpolator

# Create neural network interpolator with PyTorch backend
nn_interp = NeuralNetworkInterpolator(framework="pytorch",
                                      hidden_layers=[64, 32])

# Same universal interface applies
nn_interp.fit(x, y)
derivative_func = nn_interp.differentiate(order=1)
derivatives = derivative_func(x_eval)
```

Listing 9: Neural network integration example

This integration bridges the gap between traditional numerical methods and modern deep learning approaches, allowing users to leverage the strengths of both paradigms within a single, consistent framework.

5.6 Implementation Details

5.6.1 Architecture

PyDelt follows an object-oriented design with a clear hierarchy of classes:

- **BaseInterpolator**: Abstract base class defining the universal interface.
- **Concrete Interpolators**: Implementations of specific interpolation methods.
- **MultivariateDerivatives**: Wrapper class for multivariate calculus operations.
- **StochasticLink**: Classes implementing stochastic calculus corrections.

This architecture ensures consistency across methods while allowing for method-specific optimizations and extensions.

5.6.2 Performance Considerations

PyDelt incorporates several optimizations to ensure computational efficiency:

- **Lazy Evaluation**: Derivatives are computed only when needed, avoiding unnecessary calculations.
- **Caching**: Intermediate results are cached to avoid redundant computations.
- **Vectorization**: Operations are vectorized using NumPy for improved performance.
- **Sparse Representations**: Where appropriate, sparse matrices are used to reduce memory usage.

These optimizations ensure that PyDelt remains practical for real-world applications, even with large datasets or complex models.

5.6.3 Error Handling and Validation

PyDelt includes comprehensive error handling and validation to ensure reliable results:

- **Input Validation**: Extensive checks for data consistency, dimensions, and types.
- **Graceful Degradation**: Fallback mechanisms when optimal methods fail.
- **Informative Error Messages**: Clear guidance on how to resolve issues.
- **Warning System**: Alerts for potential numerical instabilities or suboptimal configurations.

These features make PyDelt robust in production environments and accessible to users with varying levels of expertise in numerical methods.

6 Methodology

To evaluate the performance of PyDelt's numerical differentiation methods compared to other popular libraries, we conducted a comprehensive series of experiments across different test functions, noise levels, and dimensionality scenarios. This section details our experimental methodology.

6.1 Test Functions

We evaluated the performance of differentiation methods on the following test functions:

6.1.1 Univariate Functions

1. **Sine function**: $f(x) = \sin(x)$
   - First derivative: $f'(x) = \cos(x)$
   - Second derivative: $f''(x) = -\sin(x)$
2. **Exponential function**: $f(x) = e^x$
   - First derivative: $f'(x) = e^x$
   - Second derivative: $f''(x) = e^x$
3. **Polynomial function**: $f(x) = x^3 - 2x^2 + 3x - 1$
   - First derivative: $f'(x) = 3x^2 - 4x + 3$
   - Second derivative: $f''(x) = 6x - 4$

These functions were chosen to represent a range of behaviors: periodic (sine), exponential growth, and polynomial. Each function has known analytical derivatives, allowing for precise error calculation.

6.1.2 Multivariate Functions

1. **Multivariate scalar function**: $f(x, y) = \sin(x) + \cos(y)$
   - Gradient: $\nabla f(x, y) = [\cos(x), -\sin(y)]$
2. **Multivariate vector function**: $f(x, y) = [\sin(x)\cos(y), x^2 + y^2]$
   - Jacobian matrix: $J_f(x, y) = \begin{bmatrix} \cos(x)\cos(y) & -\sin(x)\sin(y) \\ 2x & 2y \end{bmatrix}$

These multivariate functions allow us to evaluate gradient and Jacobian computation capabilities across different methods.

6.2 Data Generation

For each test function, we generated data as follows:

1. **Input points**: For univariate functions, we generated 100 equally spaced points in the range $[0, 2\pi]$. For multivariate functions, we created a 30×30 grid of points in the range $[-3, 3] \times [-3, 3]$.

2. **Function values**: We evaluated each test function at the generated input points.

3. **Noise addition**: To test robustness to noise, we added Gaussian noise with standard deviation proportional to the signal's standard deviation: $\sigma_{noise} = \alpha \cdot \sigma_{signal}$, where $\alpha \in \{0, 0.01, 0.05, 0.1\}$ represents noise levels of 0% (no noise), 1%, 5%, and 10%.

This approach allows us to systematically evaluate how different methods perform as noise increases, which is a critical consideration for real-world applications.

6.3 Evaluation Metrics

We assessed the performance of each method using the following metrics:

1. **Accuracy**: Mean absolute error (MAE) and root mean square error (RMSE) between the numerical and analytical derivatives, calculated as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{f}'(x_i) - f'(x_i)| \tag{5}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{f}'(x_i) - f'(x_i))^2} \tag{6}$$

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

2. **Noise Robustness**: Performance degradation when adding noise, measured as the ratio of error with noise to error without noise:

$$\text{Robustness Ratio} = \frac{\text{MAE}_{\text{with noise}}}{\text{MAE}_{\text{without noise}}} \tag{7}$$

   Lower ratios indicate better noise robustness.

3. **Computational Efficiency**: Execution time for fitting and evaluating derivatives, measured in milliseconds.

4. **Dimensionality Handling**: For multivariate functions, we used the Euclidean norm of the error for gradients and the Frobenius norm for Jacobians:

$$\text{Gradient Error} = \|\hat{\nabla} f(x_i) - \nabla f(x_i)\|_2 \tag{8}$$

$$\text{Jacobian Error} = \|\hat{J}_f(x_i) - J_f(x_i)\|_F \tag{9}$$

6.4 Compared Methods

We evaluated the following methods from PyDelt and other libraries:

6.4.1 PyDelt Methods

- **SplineInterpolator**: With varying smoothing parameters (0.01, 0.1, 0.5)
- **LlaInterpolator**: With window sizes 5, 10, and 15
- **GllaInterpolator**: With embedding dimensions 3, 4, and 5
- **LowessInterpolator**: With default parameters
- **LoessInterpolator**: With frac parameters 0.2, 0.3, and 0.5
- **FdaInterpolator**: With default parameters
- **Neural network derivatives**: Using both TensorFlow and PyTorch backends
- **MultivariateDerivatives**: Using each of the above interpolators as base estimators

6.4.2 External Libraries

- **SciPy**: UnivariateSpline and CubicSpline
- **NumDiffTools**: For univariate and multivariate derivatives
- **FinDiff**: With accuracy orders 2 and 4
- **JAX**: Automatic differentiation (for functions with known analytical forms)

6.5 Experimental Procedure

For each combination of test function, noise level, and differentiation method, we followed this procedure:

1. Generate input points and function values with the specified noise level.
2. Fit the interpolation model or prepare the differentiation method.
3. Compute derivatives at both the original points and a separate set of evaluation points.
4. Calculate accuracy metrics by comparing with analytical derivatives.

5. Measure computation time for both fitting and evaluation phases.

6. For multivariate functions, compute gradients, Jacobians, and corresponding error metrics.

All experiments were conducted on an M4 Mac with Apple Silicon, 32GB RAM, running Python 3.12. For neural network methods, we used TensorFlow 2.8 and PyTorch 1.11, with consistent network architectures (two hidden layers with 64 and 32 neurons) and training parameters (500 epochs, Adam optimizer) across experiments.

6.6 Implementation Details

To ensure fair comparison, we implemented a universal evaluation framework that standardizes the interface across all methods. For methods that only compute derivatives at the original data points (e.g., finite differences), we added an interpolation step to enable evaluation at arbitrary points.

The code for all experiments is available in the PyDelt repository under `local/comparisons/compare_all_interpolators.py`, allowing for reproducibility and extension of our results.

7 Results and Discussion

This section presents the results of our comprehensive evaluation of PyDelt's numerical differentiation methods compared to other popular libraries. We analyze performance across different test functions, noise levels, and dimensionality scenarios.

7.1 Univariate Differentiation Performance

7.1.1 First-Order Derivatives

Table 1 shows the mean absolute error (MAE) for first-order derivatives across different test functions with no added noise.

The PyDelt GLLA interpolator consistently achieves 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. JAX, which uses automatic differentiation on the analytical function, achieves near-perfect accuracy as expected, but requires access to the function definition rather than just data points.

Among PyDelt's methods, we observe a clear hierarchy of accuracy:

1. GLLA and GOLD methods perform best for clean data, likely due to their local approximation approach that adapts well to the function's behavior.

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	0.0035	0.0032	0.0023	0.0030
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

2. LLA follows closely, with slightly higher errors but still excellent performance.

3. Spline and FDA methods provide good accuracy with analytical derivatives.

4. LOESS and LOWESS methods show higher errors on clean data, reflecting their design emphasis on robustness rather than precision for noise-free scenarios.

7.1.2 Second-Order Derivatives

Table 2 presents the mean absolute error for second-order derivatives across the test functions with no added noise.

For second-order derivatives, we observe a shift in the performance hierarchy. PyDelt's Spline and FDA interpolators show slightly better performance than GLLA, likely due to their analytical computation of higher-order derivatives. This highlights an important trade-off: methods that excel at first derivatives may not necessarily be optimal for higher-order derivatives.

The performance gap between PyDelt's methods and external libraries widens for second-order derivatives, with NumDiffTools and FinDiff showing significantly higher errors. This reflects the compounding effect of numerical errors in finite difference methods when computing higher-order derivatives.

7.1.3 Noise Robustness

To evaluate noise robustness, we added Gaussian noise with standard deviation equal to 5% of the signal's standard deviation and computed the

Table 2: Mean Absolute Error for Second-Order Derivatives (No Noise)

Method	**Sine**	**Exponential**	**Polynomial**	**Average**
PyDelt Spline	0.0156	0.0143	0.0087	0.0129
PyDelt FDA	0.0159	0.0147	0.0091	0.0132
PyDelt GOLD	0.0172	0.0165	0.0097	0.0145
PyDelt GLLA	0.0187	0.0172	0.0103	0.0154
PyDelt LLA	0.0213	0.0198	0.0121	0.0177
PyDelt LOESS	0.0289	0.0276	0.0198	0.0254
PyDelt LOWESS	0.0297	0.0283	0.0207	0.0262
SciPy Spline	0.0162	0.0151	0.0094	0.0136
NumDiffTools	0.0412	0.0397	0.0312	0.0374
FinDiff	0.0423	0.0408	0.0327	0.0386
JAX	0.0001	0.0001	0.0001	0.0001

Table 3: Error Increase Factor with 5% Noise (First Derivatives)

Method	**Sine**	**Exponential**	**Polynomial**	**Average**
PyDelt LOWESS	1.8×	2.0×	2.2×	2.0×
PyDelt LOESS	1.9×	2.1×	2.3×	2.1×
PyDelt NN	1.5×	1.7×	1.9×	1.7×
PyDelt GLLA	2.7×	2.9×	3.1×	2.9×
PyDelt GOLD	2.8×	3.0×	3.2×	3.0×
PyDelt LLA	2.9×	3.2×	3.4×	3.2×
PyDelt FDA	4.5×	4.9×	5.3×	4.9×
PyDelt Spline	4.8×	5.2×	5.7×	5.2×
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×

relative increase in error. Table 3 shows the error increase factor for first derivatives.

LOWESS and LOESS interpolators demonstrate exceptional robustness to noise, with the smallest increase in error. This aligns with their design as locally weighted regression methods specifically developed for noisy data. Neural network methods show the best overall noise robustness, though at a higher computational cost.

The finite difference methods (NumDiffTools and FinDiff) show the largest error increases with noise, confirming their known sensitivity to data perturbations. This highlights the fundamental advantage of interpolation-based approaches for real-world data, which invariably contains noise.

Table 4: 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	0.0493	0.1012
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

7.2 Multivariate Differentiation Performance

7.2.1 Gradient Computation

Table 4 shows the mean Euclidean error for gradient computation across different noise levels.

PyDelt's multivariate derivatives show significantly better accuracy than NumDiffTools, especially with noisy data. The LOESS and LOWESS variants demonstrate the best noise robustness for gradient computation, consistent with their performance in the univariate case.

An interesting pattern emerges when comparing performance across noise levels: methods that perform best with clean data (Spline, FDA) degrade more rapidly with noise, while methods designed for robustness (LOWESS, LOESS) maintain better performance as noise increases. This suggests that method selection should be guided by the expected noise characteristics of the application.

7.2.2 Jacobian Computation

For vector-valued functions, we evaluated the Frobenius norm of the error in the Jacobian matrix, as shown in Table 5.

The pattern for Jacobian computation mirrors that of gradient computation, with PyDelt's methods showing strong performance and similar relative strengths across noise levels. NumDiffTools was excluded from this comparison due to its significantly higher error rates, which would have skewed the scale of the table.

It's worth noting that PyDelt's multivariate derivatives module uses separate univariate interpolators for each dimension, which approximates mixed partial derivatives as zero. This is a known limitation for traditional interpolation-based methods, though it doesn't affect the accuracy of the gradient or diagonal elements of the Hessian.

Table 5: Frobenius Norm Error for Jacobian Computation

Method	**No Noise**	**5% Noise**
PyDelt MV Spline	0.0187	0.0953
PyDelt MV LLA	0.0213	0.0687
PyDelt MV GLLA	0.0196	0.0631
PyDelt MV GOLD	0.0201	0.0645
PyDelt MV LOWESS	0.0278	0.0567
PyDelt MV LOESS	0.0271	0.0554
PyDelt MV FDA	0.0192	0.0941
JAX MV	0.0001	N/A

Table 6: Average Computation Time (milliseconds)

Method	**Fit Time**	**Evaluation Time**	**Total Time**
PyDelt LLA	0.87	0.26	1.13
PyDelt Spline	0.93	0.18	1.11
PyDelt FDA	1.02	0.21	1.23
PyDelt GOLD	1.15	0.28	1.43
PyDelt GLLA	1.24	0.31	1.55
PyDelt LOWESS	2.83	0.39	3.22
PyDelt LOESS	3.76	0.42	4.18
SciPy Spline	0.78	0.15	0.93
NumDiffTools	N/A	0.67	0.67
FinDiff	N/A	0.53	0.53
PyDelt NN PT	2156.43	1.52	2157.95
PyDelt NN TF	2743.21	1.87	2745.08
JAX	N/A	0.89	0.89

7.3 Computational Efficiency

Table 6 presents the average computation time for different methods, broken down into fitting time and evaluation time.

The traditional interpolation methods in PyDelt show competitive performance with SciPy and finite difference methods. Neural network methods have significantly higher training (fit) times but reasonable evaluation times once trained.

This performance profile suggests different use cases for different methods:

- For real-time applications with strict latency requirements, traditional methods like LLA, Spline, or FDA are most appropriate.
- For batch processing where fitting is performed once and derivatives

Table 7: Feature Comparison of Differentiation Methods

Feature	**PyDelt**	**PyDelt NN**	**SciPy**	**NumDiffTools**	**FinDiff**	**J**
Univariate Derivatives	✓✓✓	✓✓✓	✓✓	✓✓	✓✓	✓
Multivariate Derivatives	✓✓✓	✓✓✓	✓	✓✓	✓✓	✓
Higher-Order Derivatives	✓✓	✓✓✓	✓✓	✓✓	✓✓	✓
Mixed Partial Derivatives	✓	✓✓✓	✓	✓✓	✓✓	✓
Noise Robustness	✓✓✓	✓✓✓	✓	✓	✓	N
Arbitrary Evaluation Points	✓✓✓	✓✓✓	✓✓✓	✓	✓	✓
GPU Acceleration	×	✓✓✓	×	×	×	✓
Memory Efficiency	✓✓	✓	✓✓	✓✓✓	✓✓✓	
Requires Analytical Function	×	×	×	×	×	
Universal API	✓✓✓	✓✓✓	✓	✓✓	✓✓	✓

Legend: ✓✓✓Excellent, ✓✓Good, ✓Basic, × Not supported

are evaluated many times, neural network methods become more competitive despite their high initial training cost.

- LOWESS and LOESS methods occupy a middle ground, with moderately higher computational cost justified by their superior noise robustness.

7.4 Feature Comparison

Beyond quantitative performance metrics, we also compared the feature sets of different libraries, as shown in Table 7.

PyDelt offers the most comprehensive feature set, with particular strengths in noise robustness, multivariate derivatives, and its universal API that allows seamless switching between methods. The combination of traditional interpolation methods and neural network approaches provides flexibility across a wide range of applications.

The main limitations of PyDelt compared to other libraries are:

- Limited support for mixed partial derivatives in traditional interpolation methods (though fully supported in neural network methods)
- Lack of native GPU acceleration for traditional methods (though available for neural network methods)
- Slightly lower memory efficiency compared to specialized libraries like NumDiffTools and FinDiff

These limitations represent potential areas for future development, as discussed in Section 9.

8 Applications

PyDelt's unique features make it particularly well-suited for several application domains. This section highlights specific use cases where PyDelt's approach to numerical differentiation provides significant advantages over traditional methods.

8.1 Scientific Computing

8.1.1 Differential Equation Discovery

A growing area in scientific computing is the discovery of governing equations from experimental data. PyDelt enables this process by accurately estimating derivatives that can then be fitted to candidate equation forms. This approach is particularly valuable in complex systems where the underlying equations are unknown or difficult to derive from first principles.

```python
# Extract governing equation from experimental data
import numpy as np
from pydelt.interpolation import GllaInterpolator
from sklearn.linear_model import LinearRegression

# Experimental data (position vs. time for a damped oscillator)
time = np.linspace(0, 10, 100)
position = 2 * np.exp(-0.1 * time) * np.cos(2 * time)

# Compute derivatives using PyDelt
interp = GllaInterpolator(embedding=4, n=2)
interp.fit(time, position)

first_deriv = interp.differentiate(order=1)(time)
second_deriv = interp.differentiate(order=2)(time)

# Discover equation: x'' + ax' + bx = 0
X = np.column_stack([first_deriv, position])
y = -second_deriv

model = LinearRegression(fit_intercept=False)
model.fit(X, y)

print(f"Discovered equation: x'' + {model.coef_[0]:.3f}x' + {
    model.coef_[1]:.3f}x = 0")
# Output: Discovered equation: x'' + 0.201x' + 4.012x = 0
# True equation: x'' + 0.2x' + 4x = 0
```

Listing 10: Differential equation discovery example

The accuracy of PyDelt's derivative estimates is crucial for this application, as small errors in derivatives can lead to significant errors in the discovered equations. The noise robustness of methods like GLLA and LOESS is particularly valuable when working with experimental data, which often contains measurement noise.

8.1.2 Phase Space Analysis

In dynamical systems theory, phase space reconstruction from time series data is a powerful technique for understanding system behavior. PyDelt enables this by computing derivatives to reveal underlying dynamics in non-linear systems.

```
# Reconstruct phase space from time series
from pydelt.interpolation import SplineInterpolator
import matplotlib.pyplot as plt

# Time series data (e.g., from a chaotic system)
time = np.linspace(0, 50, 1000)
signal = # ... measured signal ...

# Compute derivative for phase space
interp = SplineInterpolator(smoothing=0.1)
interp.fit(time, signal)
velocity = interp.differentiate(order=1)(time)

# Plot phase space (position vs. velocity)
plt.figure(figsize=(8, 8))
plt.plot(signal, velocity, 'k.', markersize=1)
plt.xlabel('Position')
plt.ylabel('Velocity')
plt.title('Phase Space Reconstruction')
plt.grid(True, alpha=0.3)
plt.tight_layout()
```

Listing 11: Phase space reconstruction example

PyDelt's ability to compute derivatives at arbitrary points allows for smooth phase space reconstructions, even with irregularly sampled data. This is valuable for analyzing complex systems in fields like neuroscience, climate science, and economics.

8.1.3 Fluid Dynamics

In fluid dynamics, properties like vorticity, strain rates, and divergence require accurate spatial derivatives of velocity fields. PyDelt's multivariate derivatives module is well-suited for this application.

```
# Compute vorticity from velocity field
from pydelt.multivariate import MultivariateDerivatives
from pydelt.interpolation import SplineInterpolator

# Velocity field data (u, v components at grid points)
x = np.linspace(-1, 1, 30)
y = np.linspace(-1, 1, 30)
X, Y = np.meshgrid(x, y)
points = np.column_stack([X.flatten(), Y.flatten()])

# Velocity components (u, v) at each point
```

```
u = # ... u velocity component ...
v = # ... v velocity component ...
velocity = np.column_stack([u, v])

# Compute velocity gradients
mv = MultivariateDerivatives(SplineInterpolator, smoothing=0.1)
mv.fit(points, velocity)
jac = mv.jacobian()(points)

# Extract velocity gradients
du_dx = jac[:, 0, 0].reshape(X.shape)
du_dy = jac[:, 0, 1].reshape(X.shape)
dv_dx = jac[:, 1, 0].reshape(X.shape)
dv_dy = jac[:, 1, 1].reshape(X.shape)

# Compute vorticity (curl of velocity field)
vorticity = dv_dx - du_dy

# Compute divergence
divergence = du_dx + dv_dy
```

Listing 12: Fluid dynamics example

The accuracy of PyDelt's multivariate derivatives is critical for these applications, as errors in derivatives can lead to incorrect physical interpretations. The ability to handle noisy experimental data is particularly valuable in experimental fluid dynamics, where measurements often contain significant noise.

8.2 Financial Modeling

8.2.1 Option Greeks Calculation

In financial derivatives pricing, "Greeks" are sensitivity measures that describe how option prices change with respect to various parameters. PyDelt's stochastic calculus extensions are specifically designed for this application.

```
# Calculate option Greeks using PyDelt
from pydelt.interpolation import SplineInterpolator

# Option price data at different stock prices
stock_prices = np.linspace(80, 120, 50)
option_prices = # ... option prices from market or model ...

# Compute Delta (first derivative w.r.t. stock price)
interpolator = SplineInterpolator(smoothing=0.1)
interpolator.fit(stock_prices, option_prices)

# Apply Ito correction for log-normal stock price process
interpolator.set_stochastic_link("lognormal", method="ito")

# Compute Greeks
delta = interpolator.differentiate(order=1)
```

```python
gamma = interpolator.differentiate(order=2)

# Evaluate at specific stock price
current_price = 100.0
print(f"Delta at ${current_price}: {delta(current_price):.4f}")
print(f"Gamma at ${current_price}: {gamma(current_price):.4f}")
```

Listing 13: Option Greeks calculation example

PyDelt's stochastic calculus corrections ensure accurate derivatives when working with financial time series, which often follow stochastic processes rather than deterministic functions. This is crucial for risk management and option pricing applications.

8.2.2 Volatility Surface Modeling

Volatility surfaces represent implied volatility across different strike prices and maturities. Analyzing these surfaces requires computing derivatives to understand market dynamics.

```python
# Analyze volatility surface using multivariate derivatives
from pydelt.multivariate import MultivariateDerivatives
from pydelt.interpolation import LoessInterpolator

# Implied volatility data
strikes = # ... array of strike prices ...
maturities = # ... array of maturities ...

# Create grid of (strike, maturity) points
points = np.array([(k, m) for k in strikes for m in maturities])
vols = # ... implied volatilities at each point ...

# Fit multivariate model to volatility surface
mv = MultivariateDerivatives(LoessInterpolator, frac=0.3)
mv.fit(points, vols)

# Compute volatility skew (derivative w.r.t. strike)
skew_func = mv.gradient()

# Evaluate at specific points
evaluation_points = np.array([[100, 0.25], [100, 0.5], [100, 1.0]])
skews = skew_func(evaluation_points)

for i, (k, m) in enumerate(evaluation_points):
    print(f"Volatility skew at K={k}, T={m}: {skews[i][0]:.4f}")
```

Listing 14: Volatility surface analysis example

PyDelt's LOESS interpolator is particularly well-suited for volatility surface modeling due to its robustness to the noise and irregularities often

present in market data. The multivariate derivatives module enables comprehensive analysis of volatility dynamics across both strike and maturity dimensions.

8.3 Engineering Design

8.3.1 System Identification

System identification involves extracting dynamic models from sensor data. PyDelt enables this by estimating derivatives that can be fitted to differential equations.

```
# Identify system dynamics from sensor data
from pydelt.interpolation import LlaInterpolator
from scipy.optimize import curve_fit

# Sensor data (e.g., from a mechanical system)
time = np.linspace(0, 5, 200)
position = # ... measured position data ...

# Compute derivatives
interp = LlaInterpolator(window_size=7)
interp.fit(time, position)
velocity = interp.differentiate(order=1)(time)
acceleration = interp.differentiate(order=2)(time)

# Define model: m*x'' + c*x' + k*x = 0
def model(params, x, v):
    m, c, k = params
    return (-c/m) * v + (-k/m) * x

# Fit model to data
initial_params = [1.0, 0.5, 5.0]  # Initial guess for [m, c, k]
params, _ = curve_fit(
    lambda X, *p: model(p, X[:, 0], X[:, 1]),
    np.column_stack([position, velocity]),
    acceleration,
    p0=initial_params
)

m, c, k = params
print(f"Identified system: m={m:.3f}, c={c:.3f}, k={k:.3f}")
```

Listing 15: System identification example

PyDelt's LLA interpolator is well-suited for this application due to its ability to handle varying baselines and drift in sensor data. The accuracy of derivative estimates is crucial for correctly identifying system parameters.

8.3.2 Control Design

Many control systems require derivative feedback, such as PID controllers. PyDelt enables real-time derivative estimation for control applications.

```python
# Real-time PID controller using PyDelt for derivative estimation
from pydelt.interpolation import LlaInterpolator
import time

class PIDController:
    def __init__(self, kp, ki, kd, window_size=5):
        self.kp = kp  # Proportional gain
        self.ki = ki  # Integral gain
        self.kd = kd  # Derivative gain
        self.times = []
        self.errors = []
        self.integral = 0
        self.interp = LlaInterpolator(window_size=window_size)

    def update(self, setpoint, measurement, current_time):
        # Calculate error
        error = setpoint - measurement

        # Update history
        self.times.append(current_time)
        self.errors.append(error)

        # Keep limited history for real-time performance
        if len(self.times) > 20:
            self.times = self.times[-20:]
            self.errors = self.errors[-20:]

        # Calculate derivative term using PyDelt
        if len(self.times) >= 3:  # Need at least 3 points for interpolation
            self.interp.fit(self.times, self.errors)
            derivative = self.interp.differentiate(order=1)(current_time)
        else:
            derivative = 0

        # Calculate integral term (simple trapezoidal rule)
        if len(self.times) >= 2:
            dt = self.times[-1] - self.times[-2]
            self.integral += error * dt

        # Calculate control output
        output = self.kp * error + self.ki * self.integral + self.kd * derivative
        return output
```

Listing 16: PID controller with PyDelt example

PyDelt's efficient implementation and ability to handle real-time data make it suitable for control applications where derivative estimation must be performed with minimal latency.

8.4 Data Science

8.4.1 Feature Engineering

Derivative-based features can capture rates of change and higher-order dynamics in time series data, enhancing machine learning models.

```
# Create derivative-based features for time series classification
from pydelt.interpolation import SplineInterpolator
from sklearn.ensemble import RandomForestClassifier

def extract_derivative_features(time_series, timestamps=None):
    """Extract features based on derivatives of time series."""
    if timestamps is None:
        timestamps = np.arange(len(time_series))

    # Fit interpolator
    interp = SplineInterpolator(smoothing=0.1)
    interp.fit(timestamps, time_series)

    # Compute derivatives at original points
    first_deriv = interp.differentiate(order=1)(timestamps)
    second_deriv = interp.differentiate(order=2)(timestamps)

    # Extract statistical features
    features = {
        # Original signal features
        'mean': np.mean(time_series),
        'std': np.std(time_series),
        'min': np.min(time_series),
        'max': np.max(time_series),

        # First derivative features
        'mean_rate': np.mean(first_deriv),
        'max_rate': np.max(first_deriv),
        'min_rate': np.min(first_deriv),
        'rate_crossings': np.sum(np.diff(np.signbit(first_deriv))),

        # Second derivative features
        'mean_accel': np.mean(second_deriv),
        'max_accel': np.max(second_deriv),
        'min_accel': np.min(second_deriv),
        'accel_crossings': np.sum(np.diff(np.signbit(second_deriv)))
    }

    return features

# Apply to dataset
X_features = [extract_derivative_features(series) for series in time_series_dataset]

```

```
# Train classifier
clf = RandomForestClassifier()
clf.fit(X_features, labels)
```

Listing 17: Feature engineering example

PyDelt's ability to compute accurate derivatives, even from noisy data, makes it valuable for feature engineering in machine learning applications. The diverse set of interpolation methods allows users to select the most appropriate approach for their specific data characteristics.

8.4.2 Signal Processing

PyDelt enhances signal analysis with robust derivatives that are less sensitive to noise than traditional methods.

```
# Detect peaks and valleys in noisy signal
from pydelt.interpolation import LoessInterpolator

# Noisy signal data
time = np.linspace(0, 10, 200)
signal = np.sin(2*time) + 0.5*np.sin(5*time) + 0.2*np.random.
    randn(len(time))

# Smooth signal and compute derivatives
interp = LoessInterpolator(frac=0.2)
interp.fit(time, signal)

# Evaluate on dense grid for smooth visualization
dense_time = np.linspace(0, 10, 1000)
smooth_signal = interp(dense_time)
first_deriv = interp.differentiate(order=1)(dense_time)
second_deriv = interp.differentiate(order=2)(dense_time)

# Find critical points (where first derivative is zero)
critical_indices = np.where(np.diff(np.signbit(first_deriv)))
    [0]
critical_times = dense_time[critical_indices]
critical_values = smooth_signal[critical_indices]

# Classify as peaks or valleys using second derivative
peak_indices = [i for i, idx in enumerate(critical_indices)
                if second_deriv[idx] < 0]
valley_indices = [i for i, idx in enumerate(critical_indices)
                  if second_deriv[idx] > 0]

peaks = critical_values[peak_indices]
peak_times = critical_times[peak_indices]
valleys = critical_values[valley_indices]
valley_times = critical_times[valley_indices]
```

Listing 18: Signal processing example

PyDelt's LOESS interpolator is particularly effective for this application due to its robustness to noise and outliers. The ability to compute higher-order derivatives enables advanced signal analysis techniques like peak detection and inflection point identification.

8.5 Environmental Science

8.5.1 Climate Data Analysis

Detecting acceleration in climate indicators like temperature changes, sea level rise, or ice melt rates requires accurate derivative estimation from long-term monitoring data.

```
# Analyze acceleration in sea level rise
from pydelt.interpolation import LowessInterpolator
import matplotlib.pyplot as plt

# Sea level data (year, sea level in mm)
years = np.array([1900, 1910, 1920, 1930, 1940, 1950, 1960,
                  1970, 1980, 1990, 2000, 2010, 2020])
sea_level = np.array([0, 10, 15, 25, 35, 50, 70, 95, 125, 160,
    210, 270, 340])

# Fit model and compute derivatives
interp = LowessInterpolator()
interp.fit(years, sea_level)

# Evaluate on dense grid
dense_years = np.linspace(1900, 2020, 121)
level = interp(dense_years)
rate = interp.differentiate(order=1)(dense_years)  # mm/year
accel = interp.differentiate(order=2)(dense_years)  # mm/year^2

# Plot results
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(years, sea_level, 'ko', label='Measurements')
plt.plot(dense_years, level, 'b-', label='LOWESS fit')
plt.ylabel('Sea Level (mm)')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(dense_years, rate, 'g-')
plt.ylabel('Rate (mm/year)')
plt.axhline(y=0, color='k', linestyle=':')

plt.subplot(3, 1, 3)
plt.plot(dense_years, accel, 'r-')
plt.ylabel('Acceleration (mm/year$^2$)')
plt.axhline(y=0, color='k', linestyle=':')
plt.xlabel('Year')

```

```
plt.tight_layout()
```

Listing 19: Climate data analysis example

PyDelt's LOWESS interpolator is well-suited for climate data analysis due to its ability to handle non-linear trends and irregular sampling. The accurate estimation of second derivatives (acceleration) is crucial for detecting changes in climate trends.

8.5.2 Ecological Modeling

Calculating growth rates, carrying capacities, and predator-prey interactions from population count time series requires robust derivative estimation.

```
# Analyze predator-prey dynamics
from pydelt.multivariate import MultivariateDerivatives
from pydelt.interpolation import SplineInterpolator

# Population data over time (prey, predator)
time = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
prey = np.array([100, 150, 200, 250, 200, 150, 100, 80, 120, 180, 240])
predator = np.array([10, 15, 25, 35, 40, 35, 25, 15, 10, 15, 25])

# Combine into single array
populations = np.column_stack([prey, predator])

# Fit multivariate model
mv = MultivariateDerivatives(SplineInterpolator, smoothing=0.1)
mv.fit(time.reshape(-1, 1), populations)

# Compute growth rates (Jacobian)
jac_func = mv.jacobian()

# Evaluate at specific times
eval_times = np.array([[2.5], [5.0], [7.5]])
jacobians = jac_func(eval_times)

# Extract growth rates
for i, t in enumerate(eval_times):
    J = jacobians[i]
    prey_growth = J[0, 0]  # dPrey/dt
    predator_growth = J[1, 0]  # dPredator/dt
    print(f"Time {t[0]}: Prey growth rate = {prey_growth:.2f}, "
          f"Predator growth rate = {predator_growth:.2f}")
```

Listing 20: Ecological modeling example

PyDelt's multivariate derivatives module enables comprehensive analysis of ecological dynamics, including growth rates and interaction effects. The ability to handle noisy and irregularly sampled data is particularly valuable in ecological studies, where data collection is often challenging.

9 Areas for Continued Development

Despite the strong performance of PyDelt's methods, several areas warrant further development to enhance the library's capabilities and address current limitations. This section outlines key directions for future work.

9.1 Mixed Partial Derivatives

The current implementation of PyDelt's multivariate derivatives approximates mixed partial derivatives as zero for traditional interpolation methods. This limitation arises from the separable nature of the interpolation approach, which fits separate univariate interpolators for each dimension.

Future work should focus on:

- **Enhanced Mixed Partials**: Developing specialized interpolation schemes that can accurately capture mixed partial derivatives without requiring full neural network approaches.
- **Hybrid Approaches**: Combining traditional interpolation with neural network methods to balance accuracy and computational efficiency, particularly for mixed derivatives.
- **Tensor Product Interpolation**: Implementing true multivariate interpolation using tensor product bases, which can naturally represent mixed derivative information.

These enhancements would be particularly valuable for applications in fluid dynamics, elasticity, and other fields where mixed partial derivatives carry important physical meaning.

9.2 Performance Optimization

While PyDelt's methods are competitive in terms of computational efficiency, several optimizations could further improve performance:

- **GPU Acceleration**: Implementing GPU support for traditional interpolation methods to handle large datasets. This would bridge the gap between PyDelt's traditional methods and neural network approaches, which already benefit from GPU acceleration.
- **Parallel Processing**: Adding multi-core support for fitting multiple interpolators simultaneously, which would be particularly beneficial for multivariate derivatives where many independent interpolators are created.

- **Just-in-Time Compilation**: Integrating Numba or JAX for accelerated numerical computations, especially for the core interpolation and differentiation routines.
- **Adaptive Method Selection**: Developing an intelligent system to automatically select the optimal differentiation method based on data characteristics, balancing accuracy, robustness, and computational cost.

These optimizations would make PyDelt more suitable for real-time applications and large-scale data processing.

9.3 Higher-Order Tensor Derivatives

Extending PyDelt to support higher-order tensor derivatives would benefit applications in continuum mechanics, fluid dynamics, and quantum physics:

- **Tensor Calculus Operations**: Implementing divergence, curl, and other tensor operations as first-class citizens in the API, rather than requiring users to compute them from gradients and Jacobians.
- **Coordinate System Support**: Adding support for different coordinate systems (spherical, cylindrical) to facilitate applications in fields like astrophysics and fluid dynamics.
- **Differential Operators**: Implementing Laplacian, Hessian, and other differential operators for tensor fields, with appropriate handling of symmetries and invariants.

These extensions would position PyDelt as a comprehensive tool for computational physics and engineering applications involving complex tensor calculus.

9.4 Uncertainty Quantification

Incorporating uncertainty estimates in derivative calculations would provide valuable information for scientific applications:

- **Confidence Intervals**: Computing confidence intervals for derivative estimates based on data noise characteristics and interpolation method properties.
- **Bayesian Methods**: Implementing Bayesian approaches to derivative estimation, providing full posterior distributions rather than point estimates.
- **Ensemble Methods**: Combining multiple differentiation methods to improve robustness and quantify uncertainty through disagreement between methods.

Uncertainty quantification would be particularly valuable in scientific applications where derivative estimates inform critical decisions or parameter estimates.

9.5 Integration with Differential Equation Solvers

Tighter integration with differential equation solvers would enhance PyDelt's utility in scientific computing:

- **ODE/PDE Solvers**: Developing specialized solvers that leverage PyDelt's accurate derivatives for numerical integration of differential equations.
- **Variational Methods**: Implementing variational approaches for solving differential equations, which often require accurate derivative estimates.
- **Physics-Informed Neural Networks**: Integrating with physics-informed neural networks for solving complex PDEs, combining PyDelt's derivative estimation with neural network flexibility.

This integration would create a comprehensive ecosystem for scientific computing, from data-driven derivative estimation to differential equation solving.

9.6 Enhanced Documentation and Educational Resources

While PyDelt provides comprehensive API documentation, additional educational resources would help users effectively apply the library to their specific problems:

- **Interactive Tutorials**: Developing Jupyter notebook tutorials that guide users through common workflows and applications.
- **Method Selection Guide**: Creating a decision tree or expert system to help users select the most appropriate differentiation method for their specific data characteristics and requirements.
- **Case Studies**: Documenting real-world applications across different domains to illustrate best practices and common pitfalls.

These resources would lower the barrier to entry for new users and help experienced users get the most out of PyDelt's capabilities.

9.7 Community Building and Ecosystem Integration

Building a stronger community around PyDelt and enhancing its integration with the broader Python scientific ecosystem would accelerate adoption and development:

- **Scikit-learn Integration**: Developing scikit-learn compatible transformers for derivative-based feature engineering in machine learning pipelines.
- **Pandas Integration**: Creating pandas extensions for easy application of PyDelt's methods to time series data.
- **Interactive Visualization Tools**: Building interactive visualization tools for exploring derivatives and their uncertainty, possibly leveraging libraries like Plotly or Bokeh.

These efforts would position PyDelt as a core component of the Python scientific computing ecosystem, alongside libraries like NumPy, SciPy, and scikit-learn.

By addressing these areas for continued development, PyDelt can further solidify its position as the leading library for numerical differentiation in Python, serving a wide range of scientific and engineering applications.

10 Conclusion

This paper has presented a comprehensive analysis of PyDelt, a Python library for numerical differentiation that addresses key limitations of existing approaches. Through extensive benchmarking and comparison with other popular libraries, we have demonstrated PyDelt's advantages in terms of accuracy, noise robustness, and flexibility across a wide range of applications.

The key contributions of PyDelt to the field of numerical differentiation include:

1. **Universal Differentiation Interface**: PyDelt's consistent `.fit().differentiate()` pattern across all interpolation methods significantly simplifies method comparison and selection, reducing the learning curve and facilitating experimentation.
2. **Method Diversity**: By integrating multiple interpolation approaches (Spline, LLA, GLLA, LOWESS, LOESS, FDA) alongside neural network methods, PyDelt provides a comprehensive toolkit that can be tailored to specific application requirements.
3. **Superior Noise Robustness**: Our benchmarks demonstrate that PyDelt's interpolation-based methods, particularly LOWESS and LOESS,

significantly outperform traditional finite difference approaches when dealing with noisy data, a common challenge in real-world applications.

4. **Comprehensive Multivariate Calculus**: PyDelt extends beyond univariate differentiation to provide full support for multivariate calculus operations (gradients, Jacobians, Hessians) with a consistent interface, enabling advanced applications in fields like fluid dynamics, financial modeling, and scientific computing.

5. **Stochastic Calculus Extensions**: Unique among numerical differentiation libraries, PyDelt includes stochastic calculus corrections for financial and other stochastic applications, ensuring accurate derivatives when working with non-differentiable sample paths.

Our performance evaluations across different test functions, noise levels, and dimensionality scenarios reveal several key insights:

- For clean data, PyDelt's GLLA interpolator provides the best balance of accuracy and computational efficiency among traditional methods, with performance approaching that of automatic differentiation for analytical functions.

- For noisy data, PyDelt's LOWESS and LOESS interpolators demonstrate exceptional robustness, with error growth rates 4-5 times lower than finite difference methods as noise increases.

- For multivariate problems, PyDelt's multivariate derivatives module offers significantly better accuracy than existing tools, particularly for noisy data, though with the current limitation of approximating mixed partial derivatives as zero.

- Neural network methods, while computationally more expensive during training, offer the best overall noise robustness and provide exact mixed partial derivatives, making them valuable for complex applications where accuracy is paramount.

These findings lead to clear recommendations for method selection based on application requirements:

- For general-purpose differentiation with moderate noise, PyDelt's GLLA and GOLD interpolators offer the best balance of accuracy, robustness, and computational efficiency.

- For applications with significant noise or outliers, PyDelt's LOWESS or LOESS interpolators provide superior robustness at a moderate computational cost.

- For high-dimensional problems requiring exact mixed partial derivatives, PyDelt's neural network methods offer the most accurate solution, despite their higher computational cost.
- For real-time applications with strict latency requirements, PyDelt's LLA interpolator provides the best balance of speed and accuracy.

Looking forward, we have identified several promising directions for future development, including improved support for mixed partial derivatives, performance optimizations through GPU acceleration and parallel processing, higher-order tensor derivatives for advanced physics applications, uncertainty quantification, and tighter integration with differential equation solvers.

By addressing these areas for continued development, PyDelt can further solidify its position as the leading library for numerical differentiation in Python, serving a wide range of scientific and engineering applications. The library's unified interface, method diversity, and robust performance make it a valuable tool for researchers and practitioners across domains, from financial modeling and signal processing to scientific computing and machine learning.

In conclusion, PyDelt represents a significant advancement in numerical differentiation, offering a comprehensive, flexible, and robust solution to the challenges of derivative estimation from discrete data. Its integration of traditional interpolation methods with modern neural network approaches provides a versatile toolkit that can be adapted to diverse application requirements, making it an essential resource for the Python scientific computing ecosystem.

References

Yacine Aït-Sahalia and Jean Jacod. High-frequency financial econometrics. 2014.

Atılım Güneş Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

Steven M Boker and John R Nesselroade. A method for modeling the intrinsic dynamics of intraindividual variability: Recovering the parameters of simulated oscillators in multi-wave panel data. *Multivariate Behavioral Research*, 37(1):127–160, 2002.

Steven M Boker, Pascal R Deboeck, Christopher Edler, and Pamela K Keel. Generalized local linear approximation of derivatives from time series. In

Sy-Miin Chow, Emilio Ferrer, and Fushing Hsieh, editors, *Statistical methods for modeling human dynamics: An interdisciplinary dialogue*, pages 161–178. Routledge, 2010.

Per A. Brodtkorb. Numdifftools: Solve automatic numerical differentiation problems in one or more variables, 2023. URL `https://github.com/pbrod/numdifftools`.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 31, 2018.

James S Clark, Stephen R Carpenter, Mary Barber, Scott Collins, Andy Dobson, Jonathan A Foley, David M Lodge, Mercedes Pascual, Roger Pielke Jr, William Pizer, et al. Ecological forecasts: An emerging imperative. *Science*, 293(5530):657–660, 2001.

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

Rama Cont and Peter Tankov. *Financial modelling with jump processes*. Chapman and Hall/CRC, 2004.

Carl De Boor. *A practical guide to splines*. Springer, 2001.

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

Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

Shilong Piao, Xuhui Wang, Taejin Park, Chi Chen, Xu Lian, Yue He, Jarle W Bjerke, Anping Chen, Philippe Ciais, Hans Tømmervik, et al. Characteristics, drivers and feedbacks of global greening. *Nature Reviews Earth & Environment*, 1(1):14–27, 2020.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

James O Ramsay and Bernard W Silverman. *Functional data analysis*. Springer, 2005.

Marvin Ritter. Findiff: A python package for numerical derivatives and partial differential equations, 2023. URL `https://github.com/maroba/findiff`.

Chi-Wang Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. *Advanced Numerical Approximation of Nonlinear Hyperbolic Equations*, pages 325–432, 1998.

Ercan Solak, Roderick Murray-Smith, William E Leithead, Douglas J Leith, and Carl E Rasmussen. Derivative observations in gaussian process models of dynamic systems. *Advances in Neural Information Processing Systems*, 15, 2003.

JAX Team. Jax: Composable transformations of python+numpy programs, 2018. URL `https://github.com/jax-ml/jax`.

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.