GPUPy:A PYTHON LIBRARY FOR GPU ACCERELATED NUMERICAL METHODS

NUMERICAL METHODS IN ENGINEERING
FINAL PROJECT

MUHAMMET KADİR GÖÇER
Number: 22220030102

MERSİN UNIVERSITY

COMPUTER ENGİNEERİNG

ADVISOR
ASST. PROF. FURKAN GÖZÜKARA

MERSİN
JUNE 2025

SUMMARY

This project aims to develop a Python-based application that leverages GPU acceleration to enhance the performance of common numerical methods used in scientific computing. The application provides a user-friendly Gradio interface, allowing users to perform operations such as linear system solving, numerical integration, interpolation, and ordinary differential equation (ODE) solving. By integrating NVIDIA's CUDA-enabled CuPy library, the project demonstrates the potential for significant speedups in computationally intensive tasks compared to traditional CPU-bound methods.
The core numerical methods are implemented using **SciPy** for robust scientific computing functionalities, with custom **GPU implementations developed using CuPy** for accelerated processing. The project also features **visualization of results using Matplotlib**, providing clear graphical representations of the computations. Furthermore, this project is envisioned as a reusable and distributable **Python package, intended to be made available on PyPI**, to facilitate broader adoption and collaboration within the scientific community. The report details the implementation of these numerical techniques, the architecture of the Gradio interface, and preliminary performance observations.

Keywords: GPU Acceleration, Numerical Methods,Comparison, CuPy, SciPy, Matplotlib, PyPI, Gradio, Integration, Interpolation, ODE Solver.

CONTENTS

https://forums.developer.nvidia.com/t/understanding-cuda-texture-2d-linear-interpolation/213924
https://wwwold.mathematik.tu-dortmund.de/lsiii/cms/papers/GoeddekeStrzodka2011a.pdf
https://docs.cupy.dev/en/stable/
https://numpy.org/doc/

https://docs.nvidia.com/cuda/

7. APPENDICES

https://github.com/KdirG/GPUPy
https://libraries.io/pypi/GPUPy

1. INTRODUCTION

This section provides an overview of the project, outlining its purpose, the driving motivation behind its development, and the specific objectives it aims to achieve.

1.1. Project Overview

This project focuses on the development of a GPU-accelerated application designed to perform various numerical methods critical in scientific and engineering disciplines. The application integrates a user-friendly graphical interface built with Gradio, allowing users to interactively engage with complex computations. Key numerical methods implemented include solving linear systems, numerical integration, interpolation (linear and cubic spline), and solving ordinary differential equations (ODEs). The primary objective is to demonstrate the substantial performance gains achievable by leveraging Graphics Processing Units (GPUs) for these computationally intensive tasks, compared to traditional Central Processing Unit (CPU) based approaches. The entire framework is designed as a reusable Python package, with future plans for PyPI distribution to enhance accessibility and collaborative development.

1.2. Motivation and Problem Statement

In many scientific and engineering fields, solving complex mathematical problems often relies on numerical methods that can be computationally demanding. As data sets grow and models become more intricate, the execution time of these methods on traditional CPUs can become a significant bottleneck, hindering research and development progress. The increasing availability and processing power of GPUs present a compelling opportunity to accelerate these computations through parallel processing. However, a user-friendly and readily accessible tool that seamlessly integrates GPU acceleration for common numerical tasks is often lacking. This project addresses this gap by providing an intuitive platform that demonstrates the practical benefits of GPU computing in numerical analysis, thereby facilitating faster and more efficient scientific exploration.

1.3. Project Goals

The main goals of this project are as follows:

- **Develop a comprehensive suite of numerical methods:** Implement robust algorithms for linear system solving, numerical integration, linear interpolation, cubic spline interpolation, and ordinary differential equation solving.
- **Integrate GPU acceleration:** Utilize the CuPy library to implement GPU-accelerated versions of these numerical methods, ensuring compatibility with NVIDIA CUDA-enabled hardware.
- **Create an intuitive user interface:** Design and implement a Gradio-based web interface that allows users to easily input parameters, select methods (CPU/GPU), and visualize results without requiring in-depth programming knowledge.

- **Demonstrate performance improvement:** Conduct a comparative analysis of execution times between CPU and GPU implementations to quantify the performance benefits of GPU acceleration for each numerical method.
- **Ensure code reusability and maintainability:** Structure the project as a modular Python package, following best practices for code organization and documentation, with a view towards future public distribution via PyPI.
- **Visualize Results:** Incorporate Matplotlib for clear and informative graphical representations of the interpolation curves, integration results, and ODE solutions.

---

2. BACKGROUND AND THEORETICAL FOUNDATION

This section provides the theoretical groundwork for the numerical methods employed in this project and introduces the concept of GPU computing, specifically focusing on the CuPy library. The overall thesis writing guidelines of Mersin University Institute of Science and Technology have been taken into consideration.

2.1. Numerical Methods

Numerical methods are techniques used to solve mathematical problems that are difficult or impossible to solve analytically. They involve approximating solutions through iterative processes or discrete calculations. This project utilizes several fundamental numerical methods for various scientific computing tasks.

2.1.1. Linear System Solving

Linear systems of equations are ubiquitous in scientific and engineering applications, representing a set of linear algebraic equations. A general linear system can be expressed in the form Ax=b, where A is the coefficient matrix, x is the vector of unknowns, and b is the constant vector. Solving such systems involves finding the values of x that satisfy the equations. Common numerical techniques for solving linear systems include direct methods (e.g., Gaussian elimination, LU decomposition) and iterative methods (e.g., Jacobi, Gauss-Seidel). The choice of method often depends on the properties of the matrix A (e.g., sparsity, size) and the desired precision.

2.1.2. Numerical Integration (Definite Integrals)

Numerical integration, also known as numerical quadrature, is the process of approximating the value of a definite integral. This is particularly useful when an analytical solution is not feasible or when integrating functions derived from experimental data. Methods like the Trapezoidal Rule, Simpson's Rule, and Monte Carlo integration approximate the area under a curve by dividing it into smaller segments and summing the areas of simpler geometric shapes. The accuracy of these methods typically increases with the number of segments used.

2.1.3. Interpolation (Linear and Cubic Spline)

Interpolation is a method of constructing new data points within the range of a discrete set of known data points. It is commonly used to estimate values between observed measurements or to smooth noisy data.

- **Linear Interpolation:** This is the simplest form of interpolation, where a straight line is drawn between two adjacent data points to estimate values in between. It is computationally inexpensive but may not accurately represent non-linear trends.
- **Cubic Spline Interpolation:** This method involves fitting a series of cubic polynomials between successive data points, ensuring that the function and its first and second derivatives are continuous at the data points (knots). This results in a smoother and generally more accurate approximation of the underlying function compared to linear interpolation, especially for complex curves. Boundary conditions (e.g., 'natural', 'clamped', 'not-a-knot') are crucial for defining the behavior of the spline at the ends of the interval.

2.1.4. Ordinary Differential Equation (ODE) Solving

Ordinary Differential Equations are mathematical equations that relate a function with its derivatives. They are fundamental in modeling dynamic systems across various fields, including physics, engineering, biology, and economics. Since analytical solutions are often not possible, numerical methods are employed to approximate the solution curve. Common numerical methods for ODEs include explicit methods like Euler's method, Runge-Kutta methods (e.g., RK45), and implicit methods like Backward Differentiation Formulas (BDF). These methods approximate the solution iteratively over small time steps, calculating the next point based on the current point and the derivative at that point.

2.2. GPU Computing with CuPy

Graphics Processing Units (GPUs) are specialized electronic circuits designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Their highly parallel architecture makes them exceptionally efficient for performing a large number of simple, repetitive calculations simultaneously, which is characteristic of many numerical methods.

2.2.1. Introduction to GPU Parallelism

GPU parallelism refers to the execution of multiple computations concurrently on a GPU's numerous processing cores. Unlike CPUs, which are optimized for sequential task processing with a few powerful cores, GPUs feature thousands of smaller, more specialized cores. This architecture allows them to perform many operations simultaneously, making them ideal for "embarrassingly parallel" problems where tasks can be divided into independent sub-tasks. Leveraging GPU parallelism can drastically reduce computation times for suitable algorithms.

2.2.2. CuPy Library for NumPy-compatible GPU Arrays

CuPy is an open-source numerical library that implements a NumPy-compatible array interface on NVIDIA GPUs. It allows developers to write code that looks very similar to NumPy code but executes on the GPU, often with minimal modifications. CuPy leverages the CUDA platform, providing high-performance GPU arrays and a wide range of functions, including universal functions, linear algebra routines, and random number generation. The primary advantage of CuPy is its seamless integration with existing NumPy workflows, enabling efficient migration of CPU-bound numerical computations to the GPU for significant speedups. By calling the .get() method on a CuPy array, the data can be explicitly converted back to a NumPy array for further processing or visualization on the CPU, such as with Matplotlib.

3. SYSTEM ARCHITECTURE AND IMPLEMENTATION

This section elaborates on the overall system architecture of the GPU-accelerated numerical methods application and details the implementation aspects of its various components. The development process adhered to the guidelines outlined in the Mersin University Institute of Science and Technology Thesis Writing Guidelines.

3.1. Overall System Design

The system is designed with a layered architecture to ensure modularity, scalability, and maintainability. At the core, it integrates robust numerical algorithms, with distinct pathways for CPU-based (utilizing NumPy and SciPy) and GPU-accelerated (utilizing CuPy) computations. This backend computational engine is exposed through a user-friendly web-based graphical interface developed using Gradio. The design allows for seamless interaction, where user inputs from the Gradio frontend trigger computations on the selected backend (CPU or GPU), with results and visualizations then presented back to the user. The project is structured as a Python package (GPUPy), facilitating clear organization and potential future distribution via PyPI.

3.2. User Interface

The user interface of the application is built with Gradio, a Python library that enables rapid creation of web-based UIs for machine learning models and data science demos. Gradio's simplicity allows for the swift development of interactive components for each numerical method.

- **Linear System Solving:** The interface provides input fields for Matrix A and Vector b, allowing users to define the system Ax=b in Python list format (e.g., [[2,1],[3,4]] and [4,7]).
- **Numerical Integration:** Users can define the function to be integrated (e.g., x**2 or np.sin(x)), specify the lower (a) and upper (b) bounds, and adjust the number of points for approximation. A checkbox allows for selecting GPU acceleration.
- **Interpolation:** Input fields are provided for known x and y values, and new x values for interpolation. Users can select between linear and cubic spline methods, along with boundary conditions for splines. A checkbox for GPU usage is also available.
- **Ordinary Differential Equation (ODE) Solving:** The interface enables users to input the differential function (e.g., -y(0) or np.array([-y[0]+y[1]])), initial conditions, and time points for the solution. Method selection (e.g., RK45, BDF) is also provided.

The Gradio interface dynamically displays results, including numerical outputs, computation times, and generated plots, ensuring an intuitive user experience.

3.3. Backend Numerical Computation

The computational backbone of the application consists of both CPU and GPU implementations for each numerical method, managed to allow for direct comparison.

3.3.1. CPU Implementations (NumPy, SciPy)

For CPU-based computations, the project leverages industry-standard Python libraries:

- **NumPy:** Used for efficient array operations, fundamental linear algebra, and general numerical manipulations.
- **SciPy:** Provides a rich collection of algorithms and mathematical tools, including scipy.linalg for linear system solving, scipy.integrate for numerical integration, and scipy.interpolate for linear (interp1d) and cubic spline (CubicSpline) interpolation. These libraries offer optimized C/Fortran implementations under the hood, ensuring high performance on the CPU.

3.3.2. GPU Implementations (CuPy)

To harness the parallel processing power of GPUs, CuPy is extensively utilized:

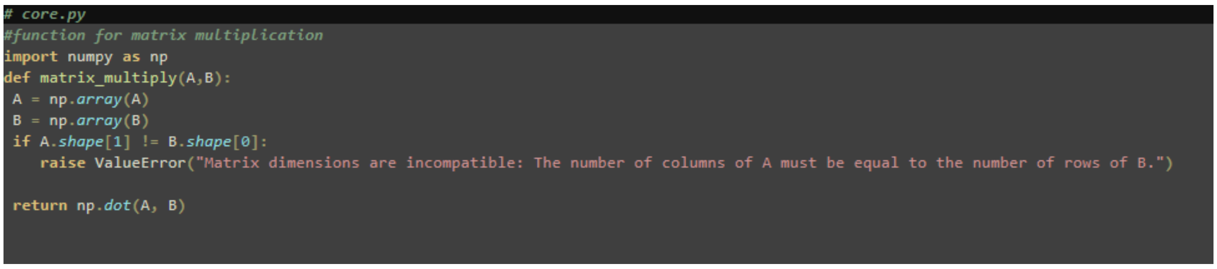
- **CuPy:** This library offers a NumPy-compatible API, allowing for relatively straightforward translation of CPU-bound NumPy/SciPy operations to the GPU. Data is transferred to the GPU as cupy.ndarray objects, computations are performed on the GPU, and results are then transferred back to the CPU as numpy.ndarray objects using cp.asnumpy() or .get() methods. This explicit conversion is crucial for compatibility with other Python libraries like Matplotlib.
- **Custom GPU Kernels :**While custom GPU kernels offer finer control, for the scope of this project, existing CuPy array operations and functions were sufficient to achieve desired accelerations, eliminating the immediate need for low-level kernel development

3.4. Module Structure

The project is organized into a well-defined module structure to enhance readability, maintainability, and reusability:

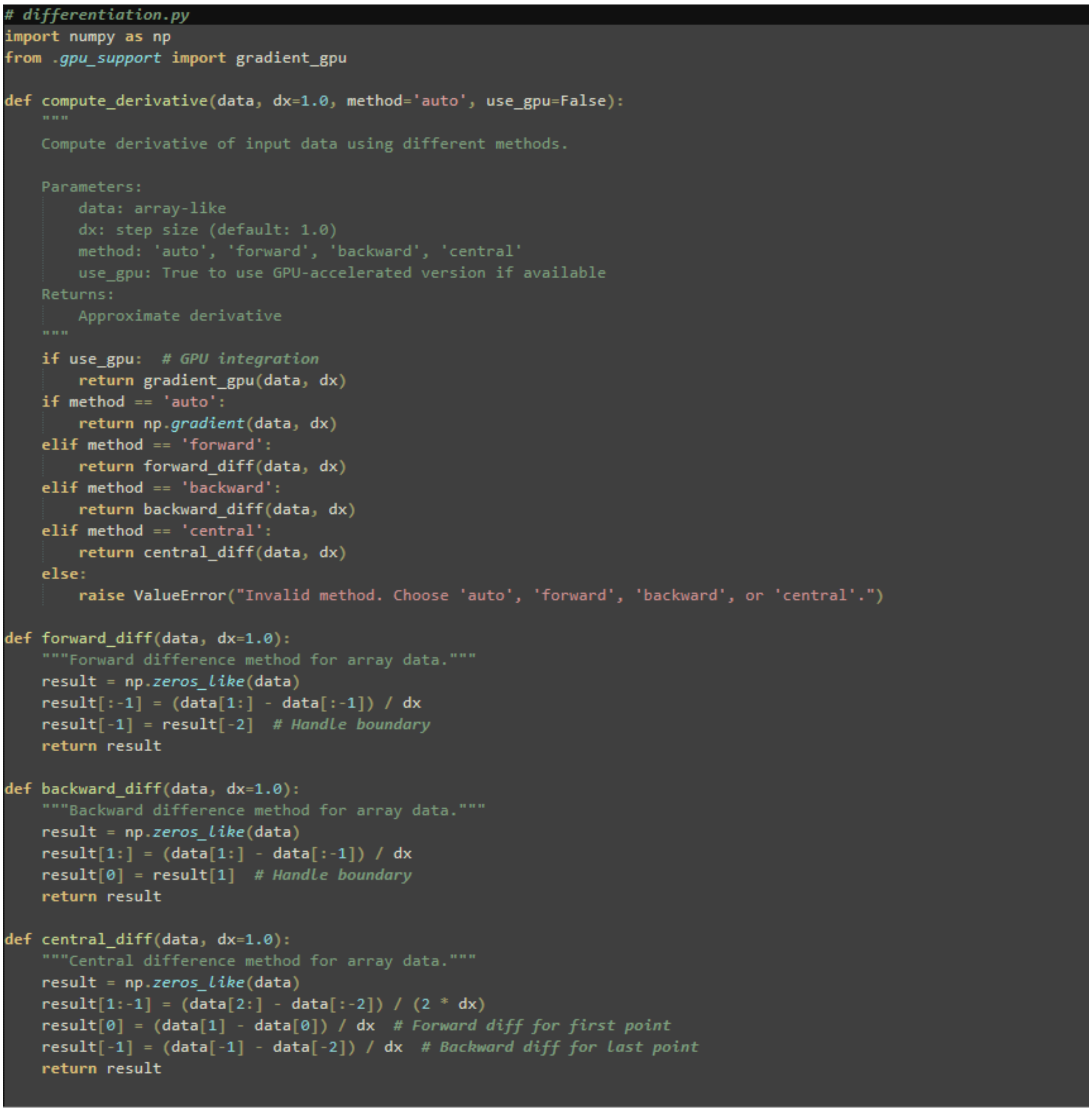
- **app.py**: This is the main script that orchestrates the Gradio application. It defines the UI layout, sets up the input/output components, and links them to the backend numerical functions.
- **GPUPy/__init__.py**: This file marks the GPUPy directory as a Python package and can be used for package-level imports or initialization.

- **GPUPy/src/numerical_methods/__init__.py**: This file aggregates imports from individual numerical method modules, allowing for cleaner imports within the GPUPy package.
- **GPUPy/src/core.py:** This module contains fundamental mathematical operations, including a robust matrix_multiply function that handles input validation and performs matrix multiplication using NumPy. It serves as a core utility for operations that might be leveraged by other numerical methods.



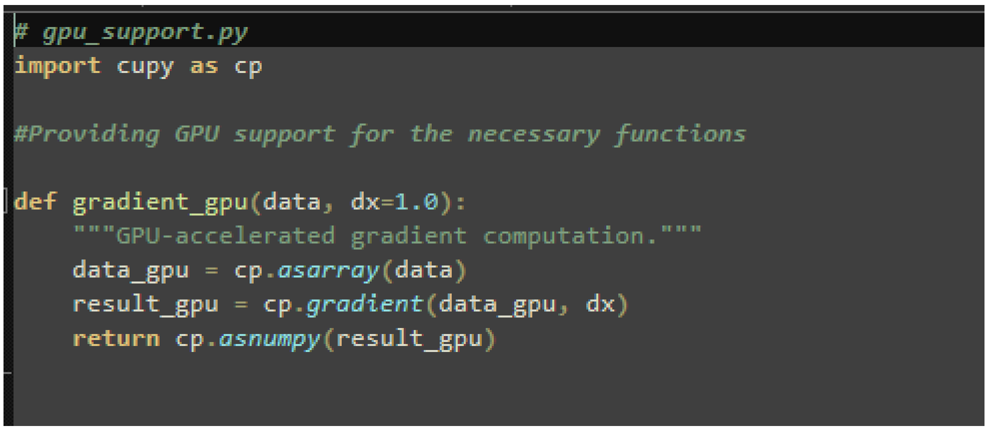
```
# core.py
#function for matrix multiplication
import numpy as np
def matrix_multiply(A,B):
 A = np.array(A)
 B = np.array(B)
 if A.shape[1] != B.shape[0]:
    raise ValueError("Matrix dimensions are incompatible: The number of columns of A must be equal to the number of rows of B.")

 return np.dot(A, B)
```

- **GPUPy/src/numerical_methods/differentiation.py**: This module provides functions for numerical differentiation, offering various finite difference methods (forward, backward, central) and includes a pathway for GPU-accelerated derivative computation using a dedicated gradient_gpu function (from gpu_support).



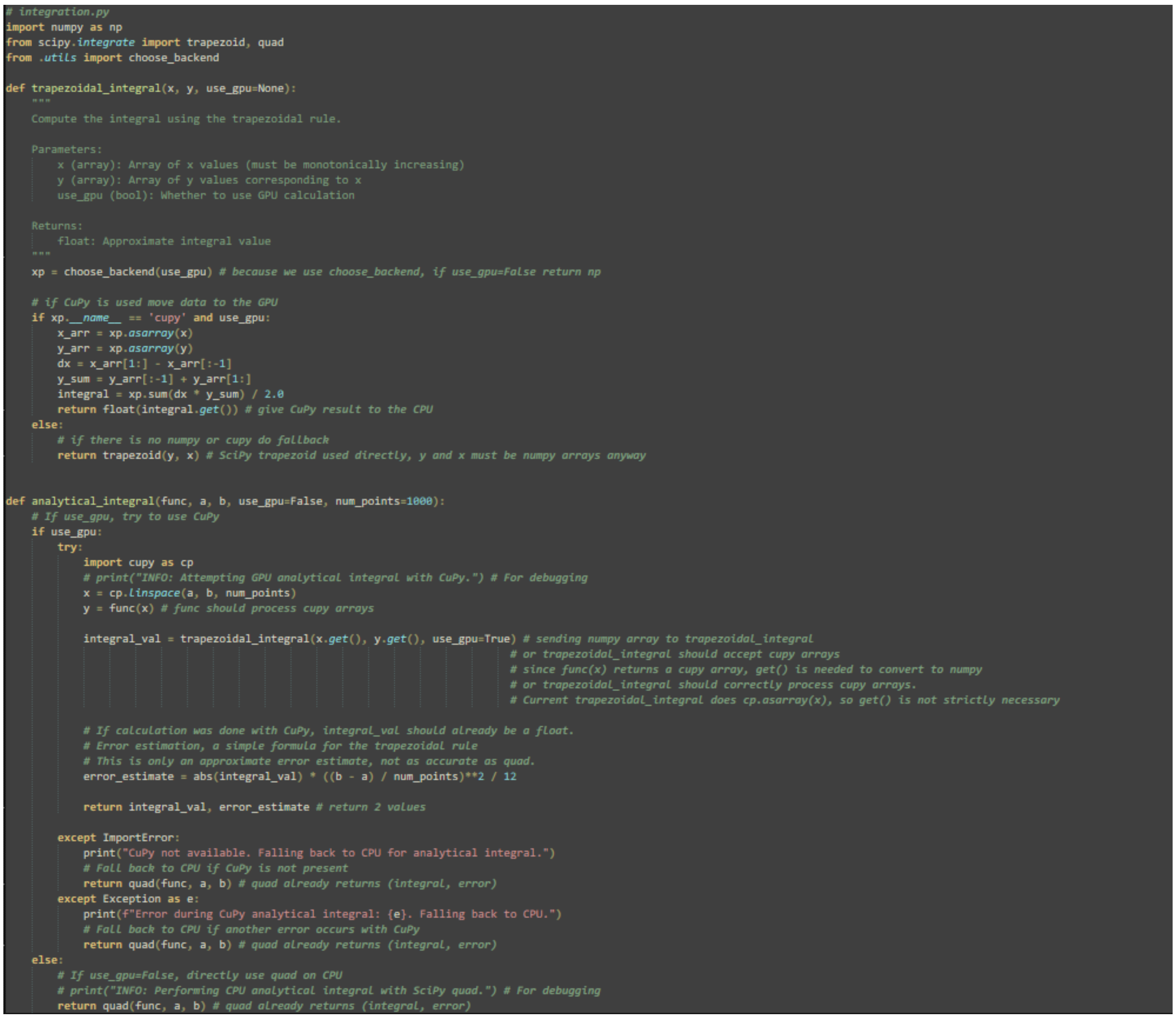
```python
# differentiation.py
import numpy as np
from .gpu_support import gradient_gpu

def compute_derivative(data, dx=1.0, method='auto', use_gpu=False):
    """
    Compute derivative of input data using different methods.

    Parameters:
        data: array-like
        dx: step size (default: 1.0)
        method: 'auto', 'forward', 'backward', 'central'
        use_gpu: True to use GPU-accelerated version if available
    Returns:
        Approximate derivative
    """
    if use_gpu:  # GPU integration
        return gradient_gpu(data, dx)
    if method == 'auto':
        return np.gradient(data, dx)
    elif method == 'forward':
        return forward_diff(data, dx)
    elif method == 'backward':
        return backward_diff(data, dx)
    elif method == 'central':
        return central_diff(data, dx)
    else:
        raise ValueError("Invalid method. Choose 'auto', 'forward', 'backward', or 'central'.")

def forward_diff(data, dx=1.0):
    """Forward difference method for array data."""
    result = np.zeros_like(data)
    result[:-1] = (data[1:] - data[:-1]) / dx
    result[-1] = result[-2]  # Handle boundary
    return result

def backward_diff(data, dx=1.0):
    """Backward difference method for array data."""
    result = np.zeros_like(data)
    result[1:] = (data[1:] - data[:-1]) / dx
    result[0] = result[1]  # Handle boundary
    return result

def central_diff(data, dx=1.0):
    """Central difference method for array data."""
    result = np.zeros_like(data)
    result[1:-1] = (data[2:] - data[:-2]) / (2 * dx)
    result[0] = (data[1] - data[0]) / dx  # Forward diff for first point
    result[-1] = (data[-1] - data[-2]) / dx  # Backward diff for last point
    return result
```

- **GPUPy/src/numerical_methods/gpu_support.py**: his module centralizes GPU-accelerated utility functions that are utilized by various numerical methods modules. It provides wrappers for CuPy operations, such as gradient_gpu, to enable seamless GPU computation and ensures results are converted back to NumPy arrays for compatibility with the CPU environment.



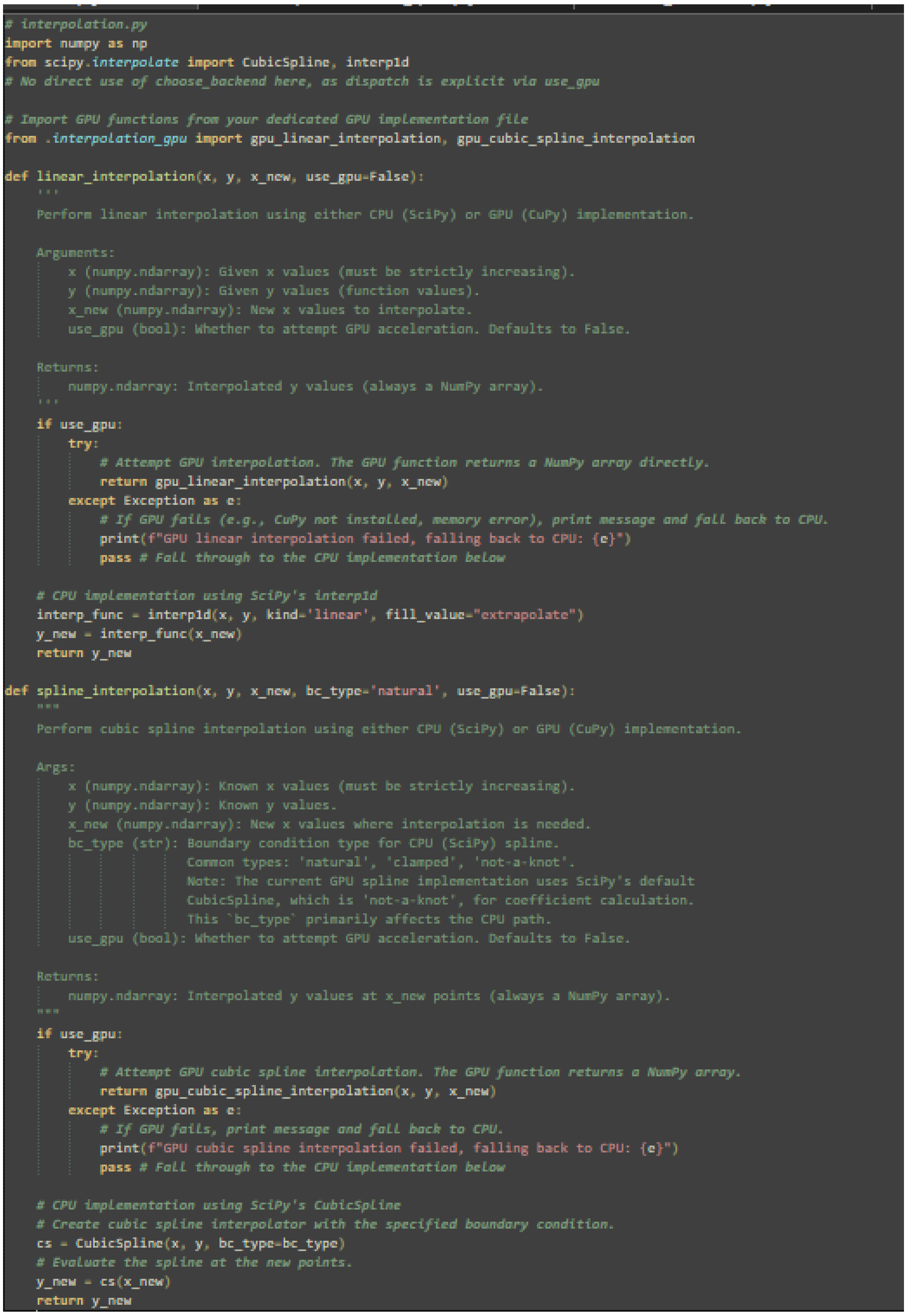
```python
# gpu_support.py
import cupy as cp

#Providing GPU support for the necessary functions

def gradient_gpu(data, dx=1.0):
    """GPU-accelerated gradient computation."""
    data_gpu = cp.asarray(data)
    result_gpu = cp.gradient(data_gpu, dx)
    return cp.asnumpy(result_gpu)
```

- **GPUPy/src/numerical_methods/integration.py**: Houses the functions for numerical integration, with separate CPU and GPU pathways.



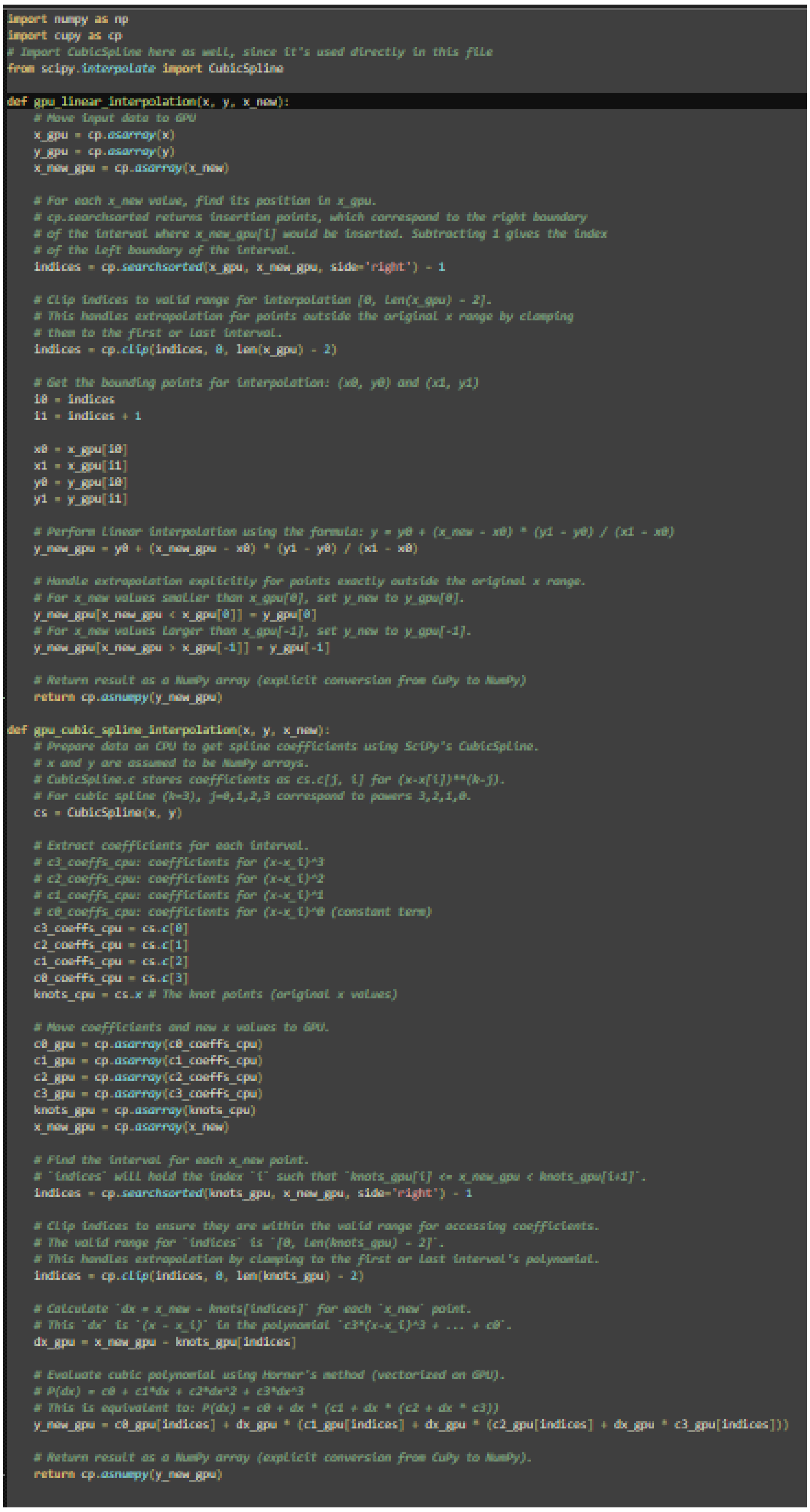
```python
# integration.py
import numpy as np
from scipy.integrate import trapezoid, quad
from .utils import choose_backend

def trapezoidal_integral(x, y, use_gpu=None):
    """
    Compute the integral using the trapezoidal rule.

    Parameters:
        x (array): Array of x values (must be monotonically increasing)
        y (array): Array of y values corresponding to x
        use_gpu (bool): Whether to use GPU calculation

    Returns:
        float: Approximate integral value
    """
    xp = choose_backend(use_gpu) # because we use choose_backend, if use_gpu=False return np

    # if CuPy is used move data to the GPU
    if xp.__name__ == 'cupy' and use_gpu:
        x_arr = xp.asarray(x)
        y_arr = xp.asarray(y)
        dx = x_arr[1:] - x_arr[:-1]
        y_sum = y_arr[:-1] + y_arr[1:]
        integral = xp.sum(dx * y_sum) / 2.0
        return float(integral.get()) # give CuPy result to the CPU
    else:
        # if there is no numpy or cupy do fallback
        return trapezoid(y, x) # SciPy trapezoid used directly, y and x must be numpy arrays anyway


def analytical_integral(func, a, b, use_gpu=False, num_points=1000):
    # If use_gpu, try to use CuPy
    if use_gpu:
        try:
            import cupy as cp
            # print("INFO: Attempting GPU analytical integral with CuPy.") # For debugging
            x = cp.linspace(a, b, num_points)
            y = func(x) # func should process cupy arrays

            integral_val = trapezoidal_integral(x.get(), y.get(), use_gpu=True) # sending numpy array to trapezoidal_integral
                                                                              # or trapezoidal_integral should accept cupy arrays
                                                                              # since func(x) returns a cupy array, get() is needed to convert to numpy
                                                                              # or trapezoidal_integral should correctly process cupy arrays.
                                                                              # Current trapezoidal_integral does cp.asarray(x), so get() is not strictly necessary

            # If calculation was done with CuPy, integral_val should already be a float.
            # Error estimation, a simple formula for the trapezoidal rule
            # This is only an approximate error estimate, not as accurate as quad.
            error_estimate = abs(integral_val) * ((b - a) / num_points)**2 / 12

            return integral_val, error_estimate # return 2 values

        except ImportError:
            print("CuPy not available. Falling back to CPU for analytical integral.")
            # Fall back to CPU if CuPy is not present
            return quad(func, a, b) # quad already returns (integral, error)
        except Exception as e:
            print(f"Error during CuPy analytical integral: {e}. Falling back to CPU.")
            # Fall back to CPU if another error occurs with CuPy
            return quad(func, a, b) # quad already returns (integral, error)
    else:
        # If use_gpu=False, directly use quad on CPU
        # print("INFO: Performing CPU analytical integral with SciPy quad.") # For debugging
        return quad(func, a, b) # quad already returns (integral, error)
```

- **GPUPy/src/numerical_methods/interpolation.py**: Provides the main entry points for linear and cubic spline interpolation, deciding whether to call CPU or GPU specific implementations based on user selection.



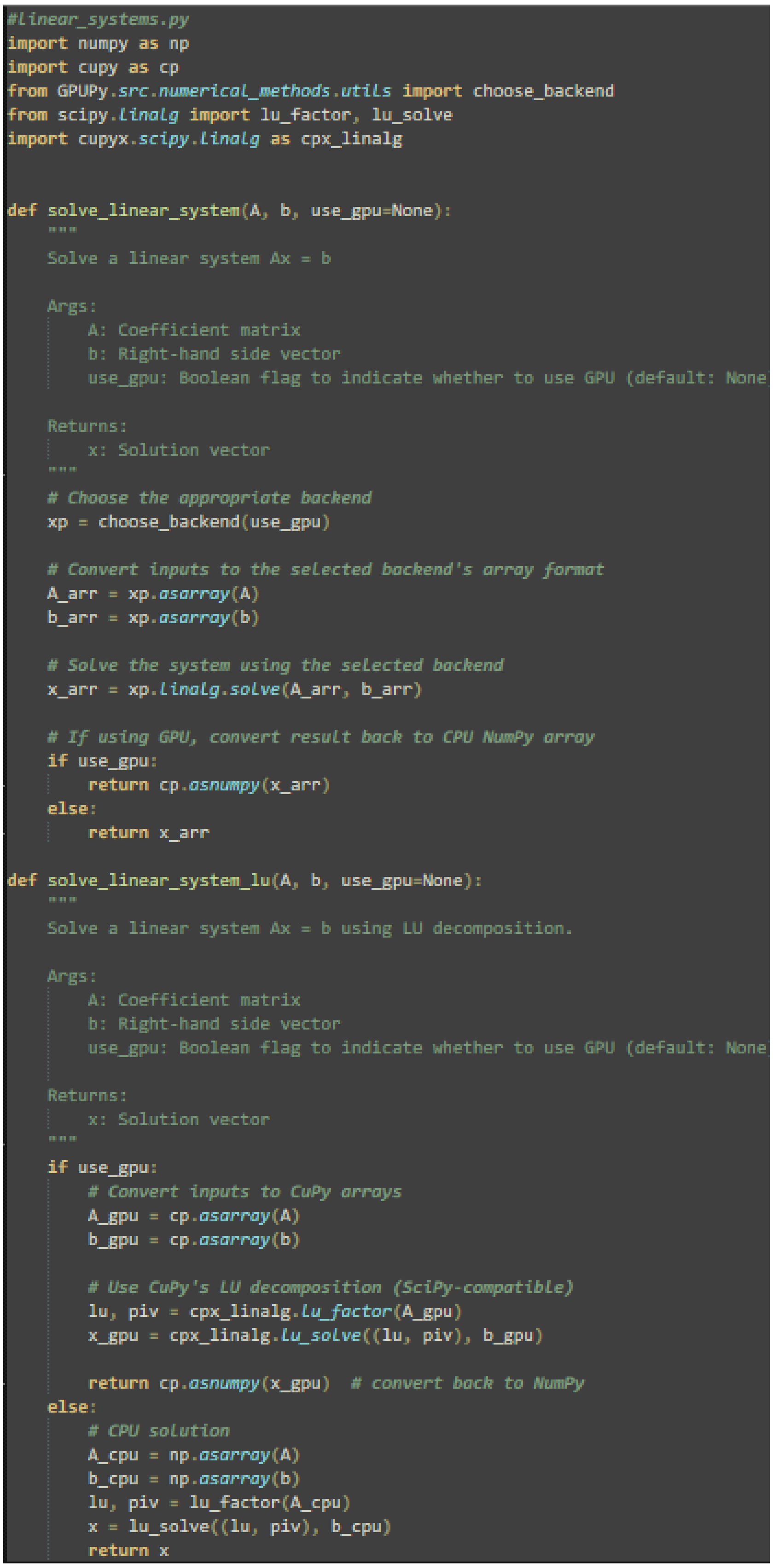
```python
# interpolation.py
import numpy as np
from scipy.interpolate import CubicSpline, interp1d
# No direct use of choose_backend here, as dispatch is explicit via use_gpu

# Import GPU functions from your dedicated GPU implementation file
from .interpolation_gpu import gpu_linear_interpolation, gpu_cubic_spline_interpolation

def linear_interpolation(x, y, x_new, use_gpu=False):
    '''
    Perform linear interpolation using either CPU (SciPy) or GPU (CuPy) implementation.

    Arguments:
        x (numpy.ndarray): Given x values (must be strictly increasing).
        y (numpy.ndarray): Given y values (function values).
        x_new (numpy.ndarray): New x values to interpolate.
        use_gpu (bool): Whether to attempt GPU acceleration. Defaults to False.

    Returns:
        numpy.ndarray: Interpolated y values (always a NumPy array).
    '''
    if use_gpu:
        try:
            # Attempt GPU interpolation. The GPU function returns a NumPy array directly.
            return gpu_linear_interpolation(x, y, x_new)
        except Exception as e:
            # If GPU fails (e.g., CuPy not installed, memory error), print message and fall back to CPU.
            print(f"GPU linear interpolation failed, falling back to CPU: {e}")
            pass # Fall through to the CPU implementation below

    # CPU implementation using SciPy's interp1d
    interp_func = interp1d(x, y, kind='linear', fill_value="extrapolate")
    y_new = interp_func(x_new)
    return y_new

def spline_interpolation(x, y, x_new, bc_type='natural', use_gpu=False):
    """
    Perform cubic spline interpolation using either CPU (SciPy) or GPU (CuPy) implementation.

    Args:
        x (numpy.ndarray): Known x values (must be strictly increasing).
        y (numpy.ndarray): Known y values.
        x_new (numpy.ndarray): New x values where interpolation is needed.
        bc_type (str): Boundary condition type for CPU (SciPy) spline.
                       Common types: 'natural', 'clamped', 'not-a-knot'.
                       Note: The current GPU spline implementation uses SciPy's default
                       CubicSpline, which is 'not-a-knot', for coefficient calculation.
                       This `bc_type` primarily affects the CPU path.
        use_gpu (bool): Whether to attempt GPU acceleration. Defaults to False.

    Returns:
        numpy.ndarray: Interpolated y values at x_new points (always a NumPy array).
    """
    if use_gpu:
        try:
            # Attempt GPU cubic spline interpolation. The GPU function returns a NumPy array.
            return gpu_cubic_spline_interpolation(x, y, x_new)
        except Exception as e:
            # If GPU fails, print message and fall back to CPU.
            print(f"GPU cubic spline interpolation failed, falling back to CPU: {e}")
            pass # Fall through to the CPU implementation below

    # CPU implementation using SciPy's CubicSpline
    # Create cubic spline interpolator with the specified boundary condition.
    cs = CubicSpline(x, y, bc_type=bc_type)
    # Evaluate the spline at the new points.
    y_new = cs(x_new)
    return y_new
```

- **GPUPy/src/numerical_methods/interpolation_gpu.py**: Contains the specific GPU-accelerated implementations for interpolation, such as gpu_linear_interpolation and gpu_cubic_spline_interpolation, ensuring cupy.ndarray outputs are converted to numpy.ndarray before returning.



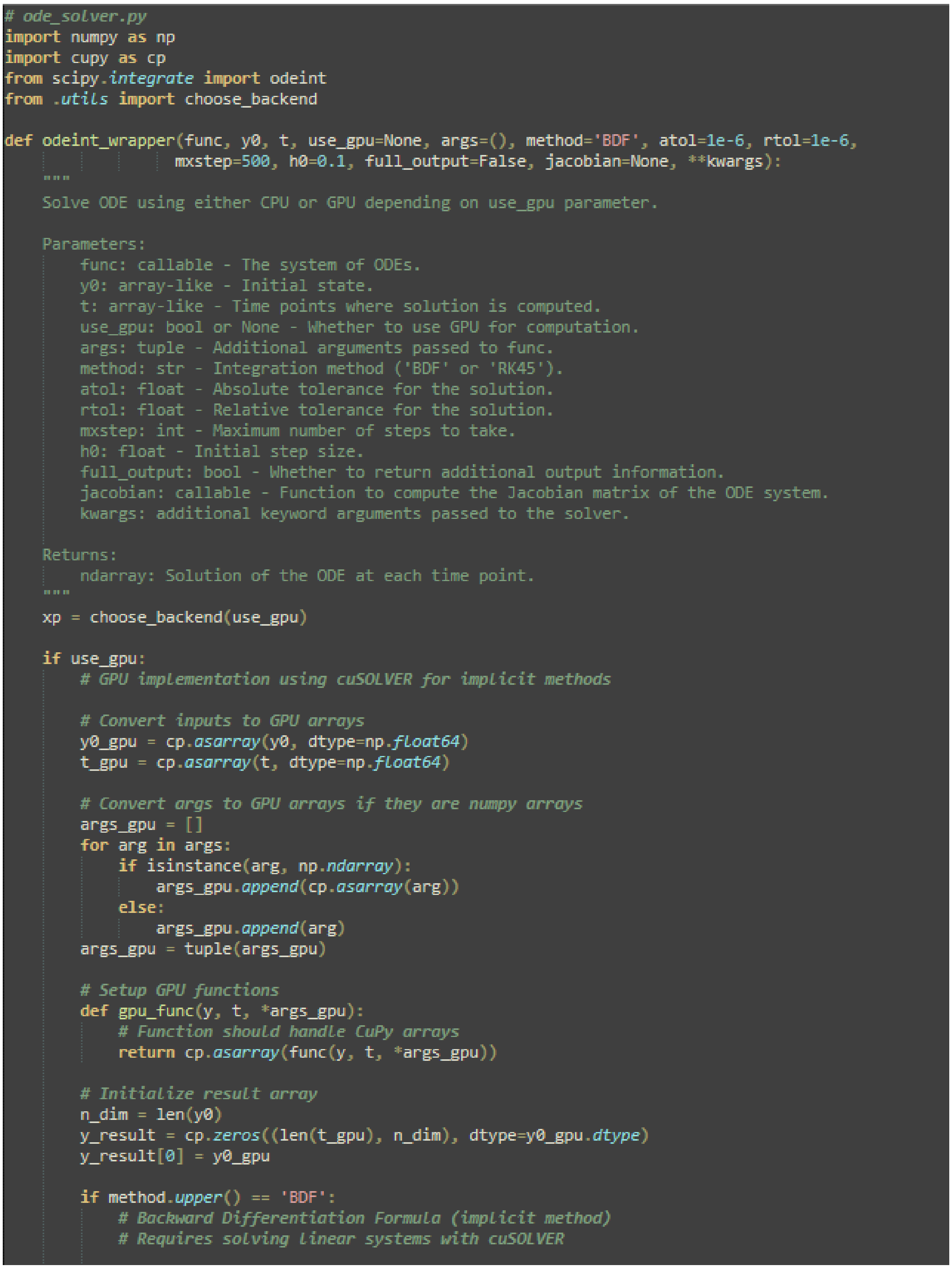
```python
import numpy as np
import cupy as cp
# Import CubicSpline here as well, since it's used directly in this file
from scipy.interpolate import CubicSpline

def gpu_linear_interpolation(x, y, x_new):
    # Move input data to GPU
    x_gpu = cp.asarray(x)
    y_gpu = cp.asarray(y)
    x_new_gpu = cp.asarray(x_new)

    # For each x_new value, find its position in x_gpu.
    # cp.searchsorted returns insertion points, which correspond to the right boundary
    # of the interval where x_new_gpu[i] would be inserted. Subtracting 1 gives the index
    # of the left boundary of the interval.
    indices = cp.searchsorted(x_gpu, x_new_gpu, side='right') - 1

    # Clip indices to valid range for interpolation [0, len(x_gpu) - 2].
    # This handles extrapolation for points outside the original x range by clamping
    # them to the first or last interval.
    indices = cp.clip(indices, 0, len(x_gpu) - 2)

    # Get the bounding points for interpolation: (x0, y0) and (x1, y1)
    i0 = indices
    i1 = indices + 1

    x0 = x_gpu[i0]
    x1 = x_gpu[i1]
    y0 = y_gpu[i0]
    y1 = y_gpu[i1]

    # Perform linear interpolation using the formula: y = y0 + (x_new - x0) * (y1 - y0) / (x1 - x0)
    y_new_gpu = y0 + (x_new_gpu - x0) * (y1 - y0) / (x1 - x0)

    # Handle extrapolation explicitly for points exactly outside the original x range.
    # For x_new values smaller than x_gpu[0], set y_new to y_gpu[0].
    y_new_gpu[x_new_gpu < x_gpu[0]] = y_gpu[0]
    # For x_new values larger than x_gpu[-1], set y_new to y_gpu[-1].
    y_new_gpu[x_new_gpu > x_gpu[-1]] = y_gpu[-1]

    # Return result as a NumPy array (explicit conversion from CuPy to NumPy)
    return cp.asnumpy(y_new_gpu)

def gpu_cubic_spline_interpolation(x, y, x_new):
    # Prepare data on CPU to get spline coefficients using SciPy's CubicSpline.
    # x and y are assumed to be NumPy arrays.
    # CubicSpline.c stores coefficients as cs.c[j, i] for (x-x[i])**(k-j).
    # For cubic spline (k=3), j=0,1,2,3 correspond to powers 3,2,1,0.
    cs = CubicSpline(x, y)

    # Extract coefficients for each interval.
    # c3_coeffs_cpu: coefficients for (x-x_i)^3
    # c2_coeffs_cpu: coefficients for (x-x_i)^2
    # c1_coeffs_cpu: coefficients for (x-x_i)^1
    # c0_coeffs_cpu: coefficients for (x-x_i)^0 (constant term)
    c3_coeffs_cpu = cs.c[0]
    c2_coeffs_cpu = cs.c[1]
    c1_coeffs_cpu = cs.c[2]
    c0_coeffs_cpu = cs.c[3]
    knots_cpu = cs.x # The knot points (original x values)

    # Move coefficients and new x values to GPU.
    c0_gpu = cp.asarray(c0_coeffs_cpu)
    c1_gpu = cp.asarray(c1_coeffs_cpu)
    c2_gpu = cp.asarray(c2_coeffs_cpu)
    c3_gpu = cp.asarray(c3_coeffs_cpu)
    knots_gpu = cp.asarray(knots_cpu)
    x_new_gpu = cp.asarray(x_new)

    # Find the interval for each x_new point.
    # `indices` will hold the index `i` such that `knots_gpu[i] <= x_new_gpu < knots_gpu[i+1]`.
    indices = cp.searchsorted(knots_gpu, x_new_gpu, side='right') - 1

    # Clip indices to ensure they are within the valid range for accessing coefficients.
    # The valid range for `indices` is `[0, len(knots_gpu) - 2]`.
    # This handles extrapolation by clamping to the first or last interval's polynomial.
    indices = cp.clip(indices, 0, len(knots_gpu) - 2)

    # Calculate `dx = x_new - knots[indices]` for each `x_new` point.
    # This `dx` is `(x - x_i)` in the polynomial `c3*(x-x_i)^3 + ... + c0`.
    dx_gpu = x_new_gpu - knots_gpu[indices]

    # Evaluate cubic polynomial using Horner's method (vectorized on GPU).
    # P(dx) = c0 + c1*dx + c2*dx^2 + c3*dx^3
    # This is equivalent to: P(dx) = c0 + dx * (c1 + dx * (c2 + dx * c3))
    y_new_gpu = c0_gpu[indices] + dx_gpu * (c1_gpu[indices] + dx_gpu * (c2_gpu[indices] + dx_gpu * c3_gpu[indices]))

    # Return result as a NumPy array (explicit conversion from CuPy to NumPy).
    return cp.asnumpy(y_new_gpu)
```

- **GPUPy/src/numerical_methods/linear_systems.py**: Contains functions related to solving linear systems, including both CPU and potentially GPU implementations.

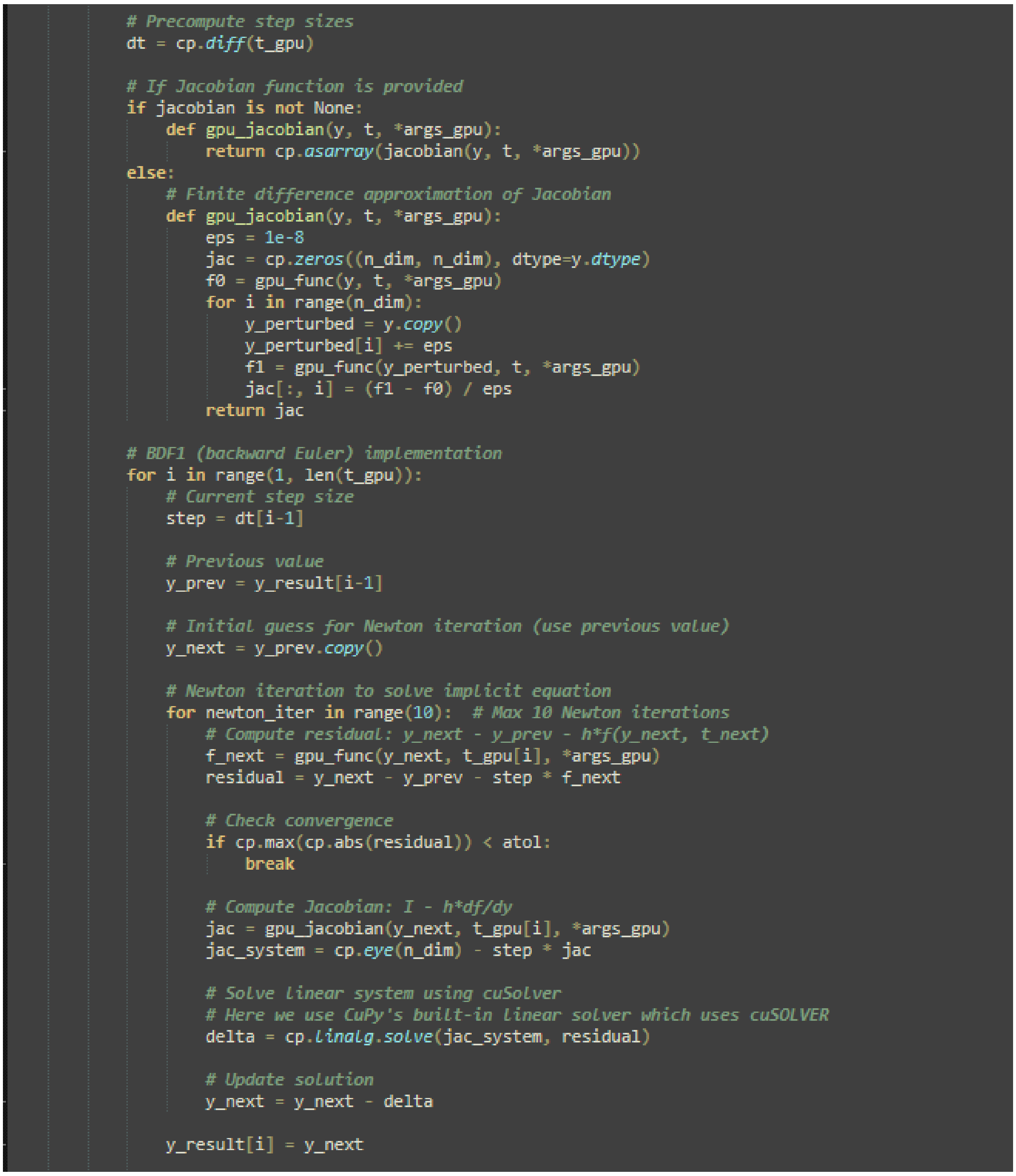


```python
#linear_systems.py
import numpy as np
import cupy as cp
from GPUPy.src.numerical_methods.utils import choose_backend
from scipy.linalg import lu_factor, lu_solve
import cupyx.scipy.linalg as cpx_linalg


def solve_linear_system(A, b, use_gpu=None):
    """
    Solve a linear system Ax = b

    Args:
        A: Coefficient matrix
        b: Right-hand side vector
        use_gpu: Boolean flag to indicate whether to use GPU (default: None

    Returns:
        x: Solution vector
    """
    # Choose the appropriate backend
    xp = choose_backend(use_gpu)

    # Convert inputs to the selected backend's array format
    A_arr = xp.asarray(A)
    b_arr = xp.asarray(b)

    # Solve the system using the selected backend
    x_arr = xp.linalg.solve(A_arr, b_arr)

    # If using GPU, convert result back to CPU NumPy array
    if use_gpu:
        return cp.asnumpy(x_arr)
    else:
        return x_arr

def solve_linear_system_lu(A, b, use_gpu=None):
    """
    Solve a linear system Ax = b using LU decomposition.

    Args:
        A: Coefficient matrix
        b: Right-hand side vector
        use_gpu: Boolean flag to indicate whether to use GPU (default: None

    Returns:
        x: Solution vector
    """
    if use_gpu:
        # Convert inputs to CuPy arrays
        A_gpu = cp.asarray(A)
        b_gpu = cp.asarray(b)

        # Use CuPy's LU decomposition (SciPy-compatible)
        lu, piv = cpx_linalg.lu_factor(A_gpu)
        x_gpu = cpx_linalg.lu_solve((lu, piv), b_gpu)

        return cp.asnumpy(x_gpu)  # convert back to NumPy
    else:
        # CPU solution
        A_cpu = np.asarray(A)
        b_cpu = np.asarray(b)
        lu, piv = lu_factor(A_cpu)
        x = lu_solve((lu, piv), b_cpu)
        return x
```

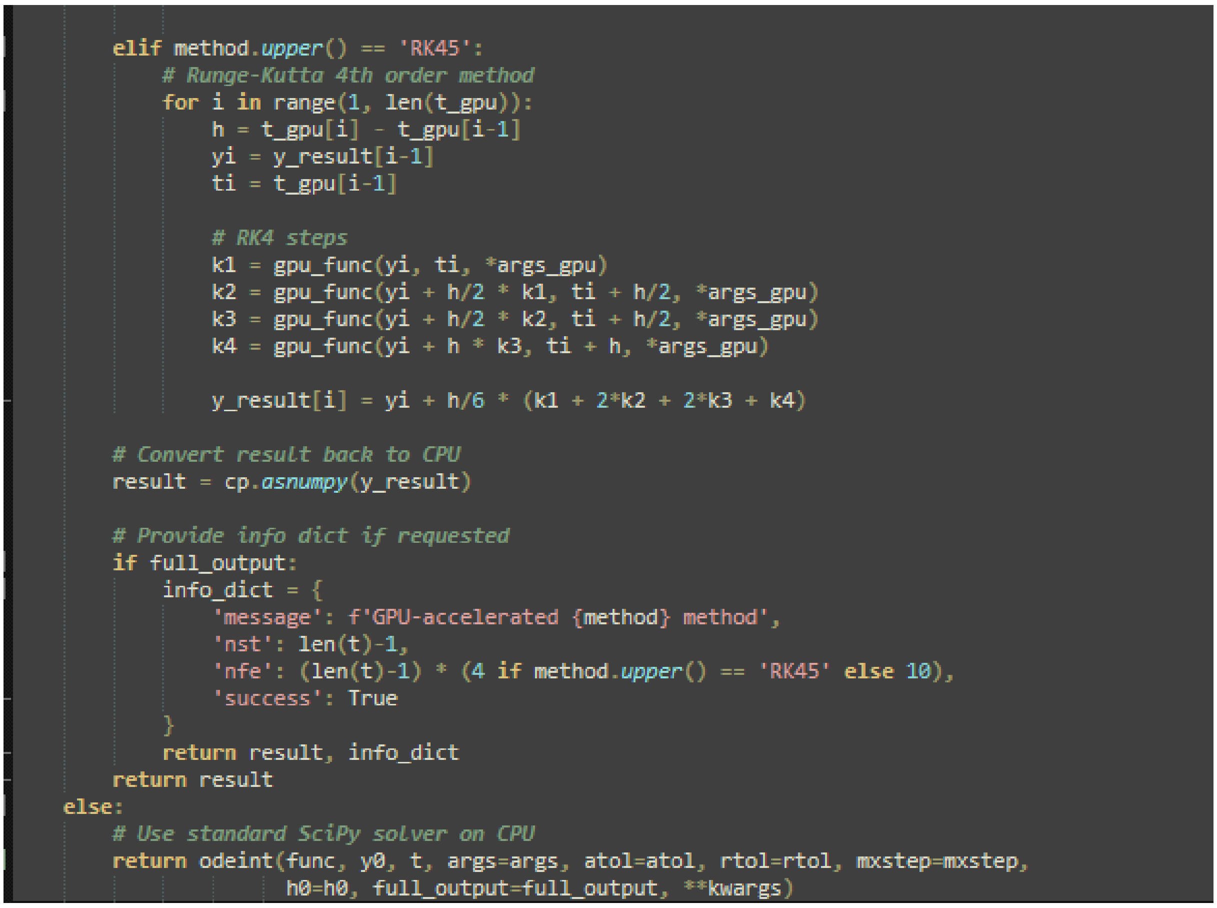
- **GPUPy/src/numerical_methods/ode_solver.py**: Implements various methods for solving ordinary differential equations, with CPU and GPU considerations.



```python
# ode_solver.py
import numpy as np
import cupy as cp
from scipy.integrate import odeint
from .utils import choose_backend

def odeint_wrapper(func, y0, t, use_gpu=None, args=(), method='BDF', atol=1e-6, rtol=1e-6,
                   mxstep=500, h0=0.1, full_output=False, jacobian=None, **kwargs):
    """
    Solve ODE using either CPU or GPU depending on use_gpu parameter.

    Parameters:
        func: callable - The system of ODEs.
        y0: array-like - Initial state.
        t: array-like - Time points where solution is computed.
        use_gpu: bool or None - Whether to use GPU for computation.
        args: tuple - Additional arguments passed to func.
        method: str - Integration method ('BDF' or 'RK45').
        atol: float - Absolute tolerance for the solution.
        rtol: float - Relative tolerance for the solution.
        mxstep: int - Maximum number of steps to take.
        h0: float - Initial step size.
        full_output: bool - Whether to return additional output information.
        jacobian: callable - Function to compute the Jacobian matrix of the ODE system.
        kwargs: additional keyword arguments passed to the solver.

    Returns:
        ndarray: Solution of the ODE at each time point.
    """
    xp = choose_backend(use_gpu)

    if use_gpu:
        # GPU implementation using cuSOLVER for implicit methods

        # Convert inputs to GPU arrays
        y0_gpu = cp.asarray(y0, dtype=np.float64)
        t_gpu = cp.asarray(t, dtype=np.float64)

        # Convert args to GPU arrays if they are numpy arrays
        args_gpu = []
        for arg in args:
            if isinstance(arg, np.ndarray):
                args_gpu.append(cp.asarray(arg))
            else:
                args_gpu.append(arg)
        args_gpu = tuple(args_gpu)

        # Setup GPU functions
        def gpu_func(y, t, *args_gpu):
            # Function should handle CuPy arrays
            return cp.asarray(func(y, t, *args_gpu))

        # Initialize result array
        n_dim = len(y0)
        y_result = cp.zeros((len(t_gpu), n_dim), dtype=y0_gpu.dtype)
        y_result[0] = y0_gpu

        if method.upper() == 'BDF':
            # Backward Differentiation Formula (implicit method)
            # Requires solving linear systems with cuSOLVER
```

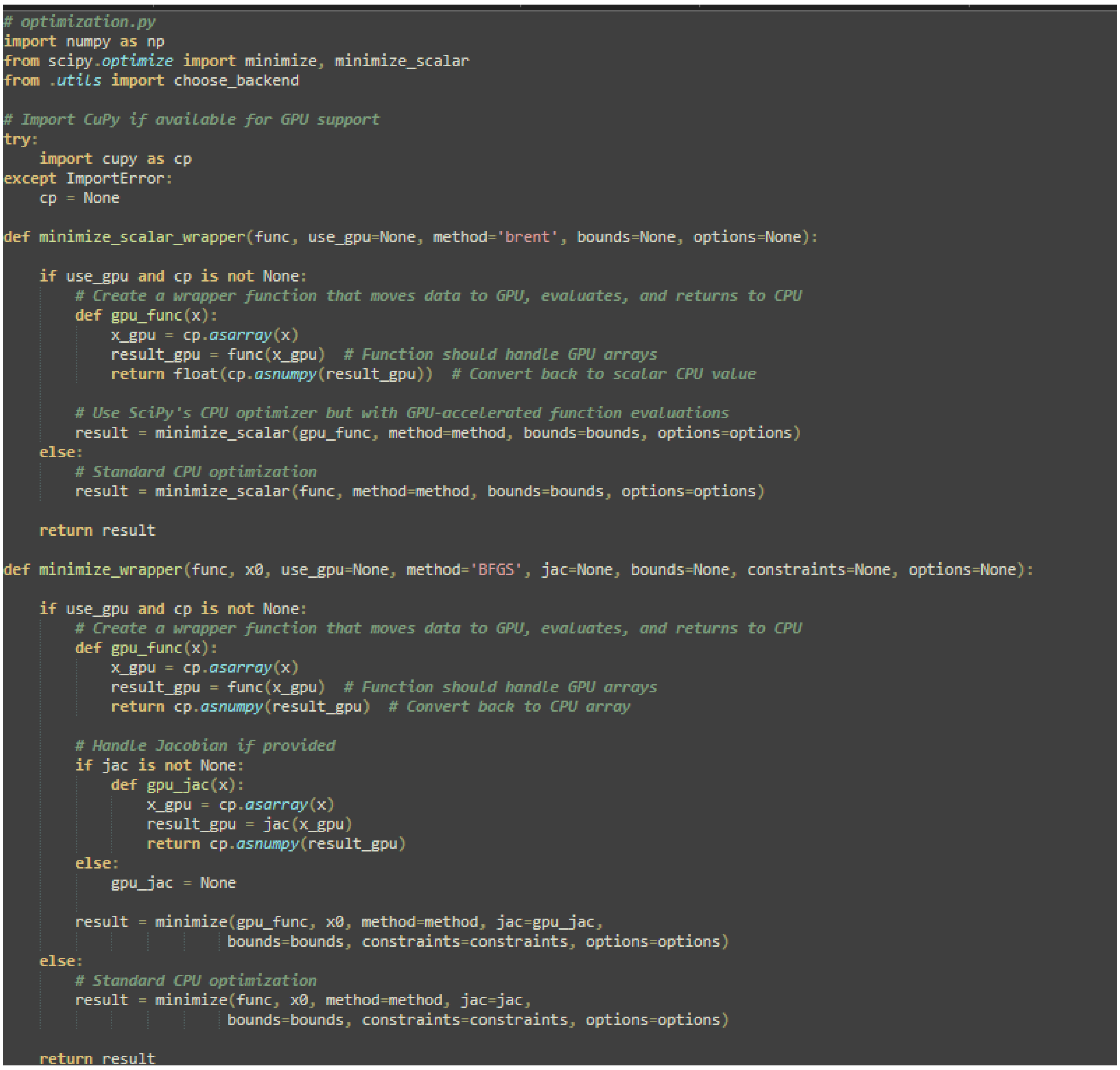


```
        # Precompute step sizes
        dt = cp.diff(t_gpu)

        # If Jacobian function is provided
        if jacobian is not None:
            def gpu_jacobian(y, t, *args_gpu):
                return cp.asarray(jacobian(y, t, *args_gpu))
        else:
            # Finite difference approximation of Jacobian
            def gpu_jacobian(y, t, *args_gpu):
                eps = 1e-8
                jac = cp.zeros((n_dim, n_dim), dtype=y.dtype)
                f0 = gpu_func(y, t, *args_gpu)
                for i in range(n_dim):
                    y_perturbed = y.copy()
                    y_perturbed[i] += eps
                    f1 = gpu_func(y_perturbed, t, *args_gpu)
                    jac[:, i] = (f1 - f0) / eps
                return jac

        # BDF1 (backward Euler) implementation
        for i in range(1, len(t_gpu)):
            # Current step size
            step = dt[i-1]

            # Previous value
            y_prev = y_result[i-1]

            # Initial guess for Newton iteration (use previous value)
            y_next = y_prev.copy()

            # Newton iteration to solve implicit equation
            for newton_iter in range(10):  # Max 10 Newton iterations
                # Compute residual: y_next - y_prev - h*f(y_next, t_next)
                f_next = gpu_func(y_next, t_gpu[i], *args_gpu)
                residual = y_next - y_prev - step * f_next

                # Check convergence
                if cp.max(cp.abs(residual)) < atol:
                    break

                # Compute Jacobian: I - h*df/dy
                jac = gpu_jacobian(y_next, t_gpu[i], *args_gpu)
                jac_system = cp.eye(n_dim) - step * jac

                # Solve linear system using cuSolver
                # Here we use CuPy's built-in linear solver which uses cuSOLVER
                delta = cp.linalg.solve(jac_system, residual)

                # Update solution
                y_next = y_next - delta

            y_result[i] = y_next
```



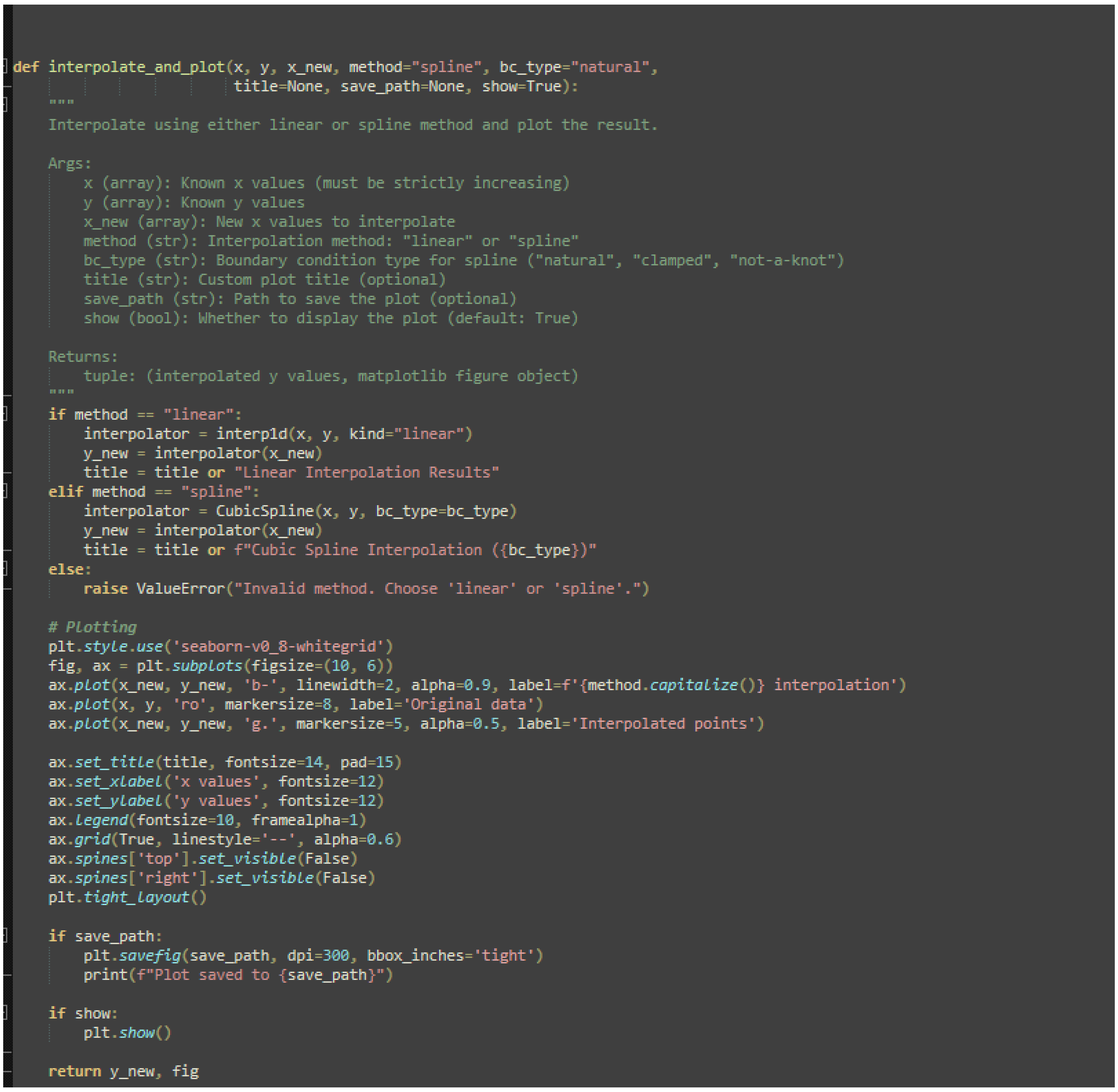
```
        elif method.upper() == 'RK45':
            # Runge-Kutta 4th order method
            for i in range(1, len(t_gpu)):
                h = t_gpu[i] - t_gpu[i-1]
                yi = y_result[i-1]
                ti = t_gpu[i-1]

                # RK4 steps
                k1 = gpu_func(yi, ti, *args_gpu)
                k2 = gpu_func(yi + h/2 * k1, ti + h/2, *args_gpu)
                k3 = gpu_func(yi + h/2 * k2, ti + h/2, *args_gpu)
                k4 = gpu_func(yi + h * k3, ti + h, *args_gpu)

                y_result[i] = yi + h/6 * (k1 + 2*k2 + 2*k3 + k4)

        # Convert result back to CPU
        result = cp.asnumpy(y_result)

        # Provide info dict if requested
        if full_output:
            info_dict = {
                'message': f'GPU-accelerated {method} method',
                'nst': len(t)-1,
                'nfe': (len(t)-1) * (4 if method.upper() == 'RK45' else 10),
                'success': True
            }
            return result, info_dict
        return result
    else:
        # Use standard SciPy solver on CPU
        return odeint(func, y0, t, args=args, atol=atol, rtol=rtol, mxstep=mxstep,
                      h0=h0, full_output=full_output, **kwargs)
```

- **GPUPy/src/numerical_methods/optimization.py**: This module implements numerical optimization routines, leveraging scipy.optimize for both scalar and multivariate function minimization. Crucially, it allows for GPU acceleration of the *objective function evaluations* by dynamically wrapping user-defined functions to handle data transfer to/from CuPy arrays on the GPU.



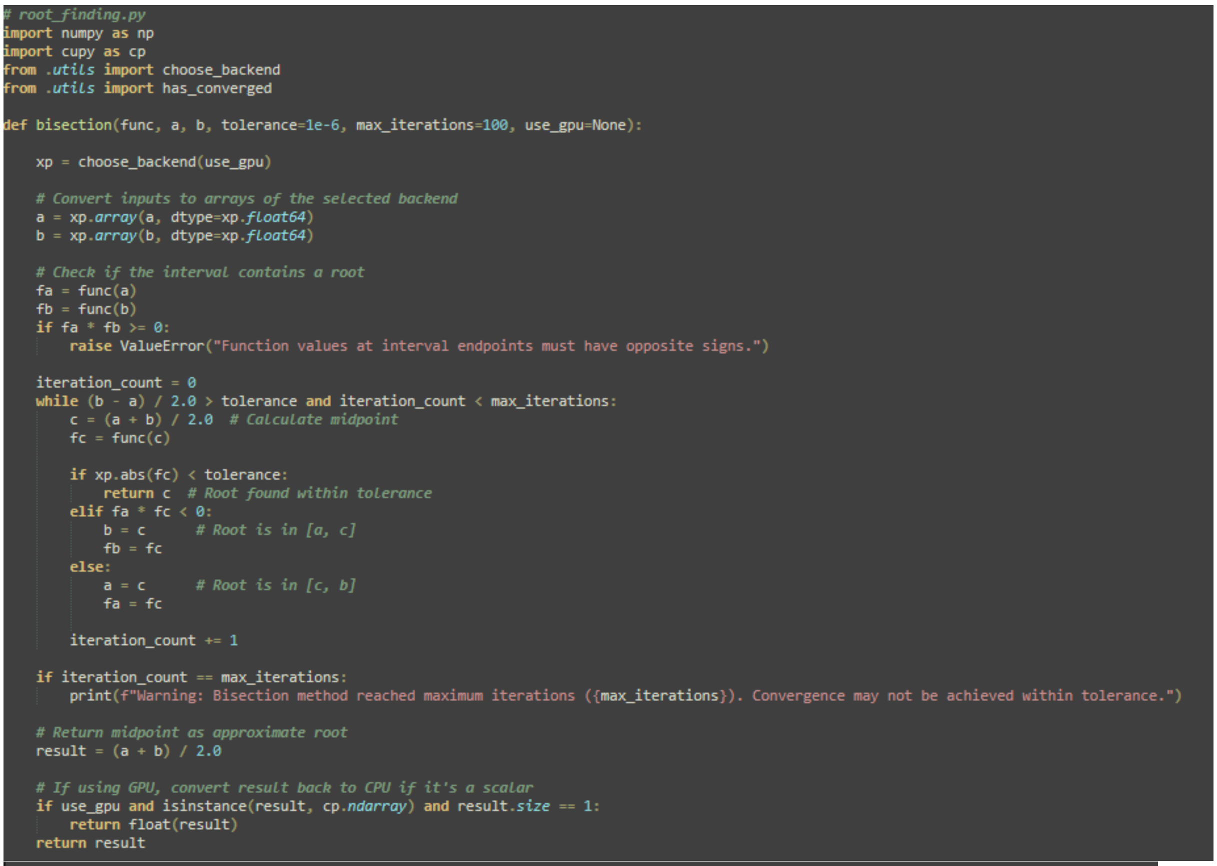
```python
# optimization.py
import numpy as np
from scipy.optimize import minimize, minimize_scalar
from .utils import choose_backend

# Import CuPy if available for GPU support
try:
    import cupy as cp
except ImportError:
    cp = None

def minimize_scalar_wrapper(func, use_gpu=None, method='brent', bounds=None, options=None):

    if use_gpu and cp is not None:
        # Create a wrapper function that moves data to GPU, evaluates, and returns to CPU
        def gpu_func(x):
            x_gpu = cp.asarray(x)
            result_gpu = func(x_gpu)  # Function should handle GPU arrays
            return float(cp.asnumpy(result_gpu))  # Convert back to scalar CPU value

        # Use SciPy's CPU optimizer but with GPU-accelerated function evaluations
        result = minimize_scalar(gpu_func, method=method, bounds=bounds, options=options)
    else:
        # Standard CPU optimization
        result = minimize_scalar(func, method=method, bounds=bounds, options=options)

    return result

def minimize_wrapper(func, x0, use_gpu=None, method='BFGS', jac=None, bounds=None, constraints=None, options=None):

    if use_gpu and cp is not None:
        # Create a wrapper function that moves data to GPU, evaluates, and returns to CPU
        def gpu_func(x):
            x_gpu = cp.asarray(x)
            result_gpu = func(x_gpu)  # Function should handle GPU arrays
            return cp.asnumpy(result_gpu)  # Convert back to CPU array

        # Handle Jacobian if provided
        if jac is not None:
            def gpu_jac(x):
                x_gpu = cp.asarray(x)
                result_gpu = jac(x_gpu)
                return cp.asnumpy(result_gpu)
        else:
            gpu_jac = None

        result = minimize(gpu_func, x0, method=method, jac=gpu_jac,
                          bounds=bounds, constraints=constraints, options=options)
    else:
        # Standard CPU optimization
        result = minimize(func, x0, method=method, jac=jac,
                          bounds=bounds, constraints=constraints, options=options)

    return result
```

- **GPUPy/src/numerical_methods/plot.py**: This module is dedicated to generating high-quality visualizations of numerical results, particularly for interpolation. It utilizes Matplotlib to create plots that display original data points, interpolated curves, and new interpolated points, enhancing the interpretability of the results presented in the Gradio interface.

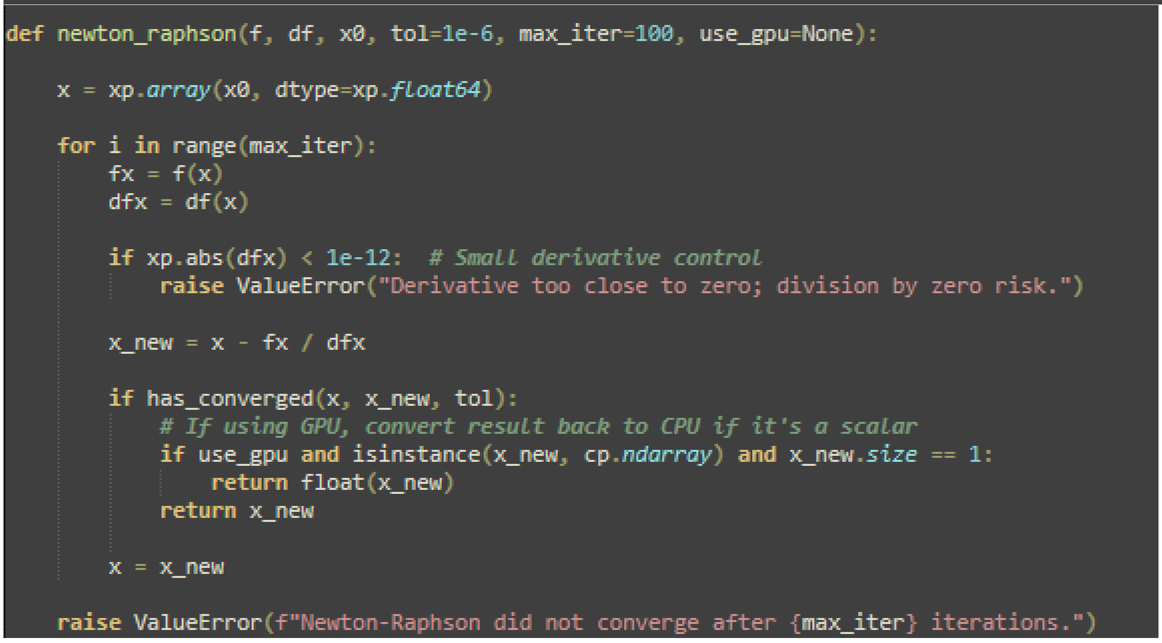


```python
def interpolate_and_plot(x, y, x_new, method="spline", bc_type="natural",
                         title=None, save_path=None, show=True):
    """
    Interpolate using either linear or spline method and plot the result.

    Args:
        x (array): Known x values (must be strictly increasing)
        y (array): Known y values
        x_new (array): New x values to interpolate
        method (str): Interpolation method: "linear" or "spline"
        bc_type (str): Boundary condition type for spline ("natural", "clamped", "not-a-knot")
        title (str): Custom plot title (optional)
        save_path (str): Path to save the plot (optional)
        show (bool): Whether to display the plot (default: True)

    Returns:
        tuple: (interpolated y values, matplotlib figure object)
    """
    if method == "linear":
        interpolator = interp1d(x, y, kind="linear")
        y_new = interpolator(x_new)
        title = title or "Linear Interpolation Results"
    elif method == "spline":
        interpolator = CubicSpline(x, y, bc_type=bc_type)
        y_new = interpolator(x_new)
        title = title or f"Cubic Spline Interpolation ({bc_type})"
    else:
        raise ValueError("Invalid method. Choose 'linear' or 'spline'.")

    # Plotting
    plt.style.use('seaborn-v0_8-whitegrid')
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.plot(x_new, y_new, 'b-', linewidth=2, alpha=0.9, label=f'{method.capitalize()} interpolation')
    ax.plot(x, y, 'ro', markersize=8, label='Original data')
    ax.plot(x_new, y_new, 'g.', markersize=5, alpha=0.5, label='Interpolated points')

    ax.set_title(title, fontsize=14, pad=15)
    ax.set_xlabel('x values', fontsize=12)
    ax.set_ylabel('y values', fontsize=12)
    ax.legend(fontsize=10, framealpha=1)
    ax.grid(True, linestyle='--', alpha=0.6)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    plt.tight_layout()

    if save_path:
        plt.savefig(save_path, dpi=300, bbox_inches='tight')
        print(f"Plot saved to {save_path}")

    if show:
        plt.show()

    return y_new, fig
```

- **GPUPy/src/numerical_methods/root_finding.py**: This module provides implementations for finding the roots of functions, including robust methods like **Bisection** and efficient iterative methods like **Newton-Raphson**. It incorporates dynamic backend selection (CPU/GPU) via utils.choose_backend and utilizes utils.has_converged for iteration termination, ensuring flexible and performant root computations.

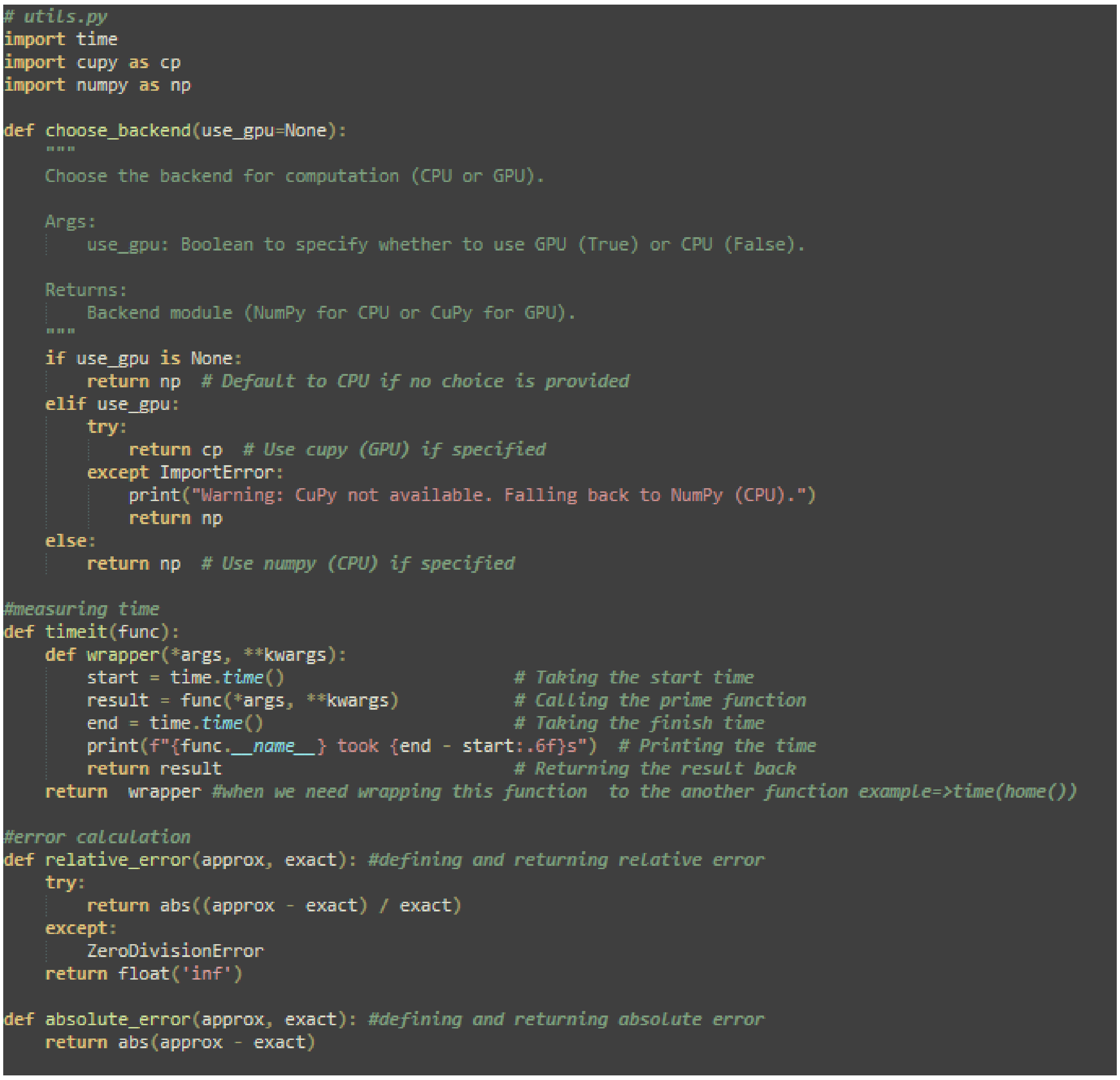


```python
# root_finding.py
import numpy as np
import cupy as cp
from .utils import choose_backend
from .utils import has_converged

def bisection(func, a, b, tolerance=1e-6, max_iterations=100, use_gpu=None):

    xp = choose_backend(use_gpu)

    # Convert inputs to arrays of the selected backend
    a = xp.array(a, dtype=xp.float64)
    b = xp.array(b, dtype=xp.float64)

    # Check if the interval contains a root
    fa = func(a)
    fb = func(b)
    if fa * fb >= 0:
        raise ValueError("Function values at interval endpoints must have opposite signs.")

    iteration_count = 0
    while (b - a) / 2.0 > tolerance and iteration_count < max_iterations:
        c = (a + b) / 2.0  # Calculate midpoint
        fc = func(c)

        if xp.abs(fc) < tolerance:
            return c  # Root found within tolerance
        elif fa * fc < 0:
            b = c      # Root is in [a, c]
            fb = fc
        else:
            a = c      # Root is in [c, b]
            fa = fc

        iteration_count += 1

    if iteration_count == max_iterations:
        print(f"Warning: Bisection method reached maximum iterations ({max_iterations}). Convergence may not be achieved within tolerance.")

    # Return midpoint as approximate root
    result = (a + b) / 2.0

    # If using GPU, convert result back to CPU if it's a scalar
    if use_gpu and isinstance(result, cp.ndarray) and result.size == 1:
        return float(result)
    return result
```

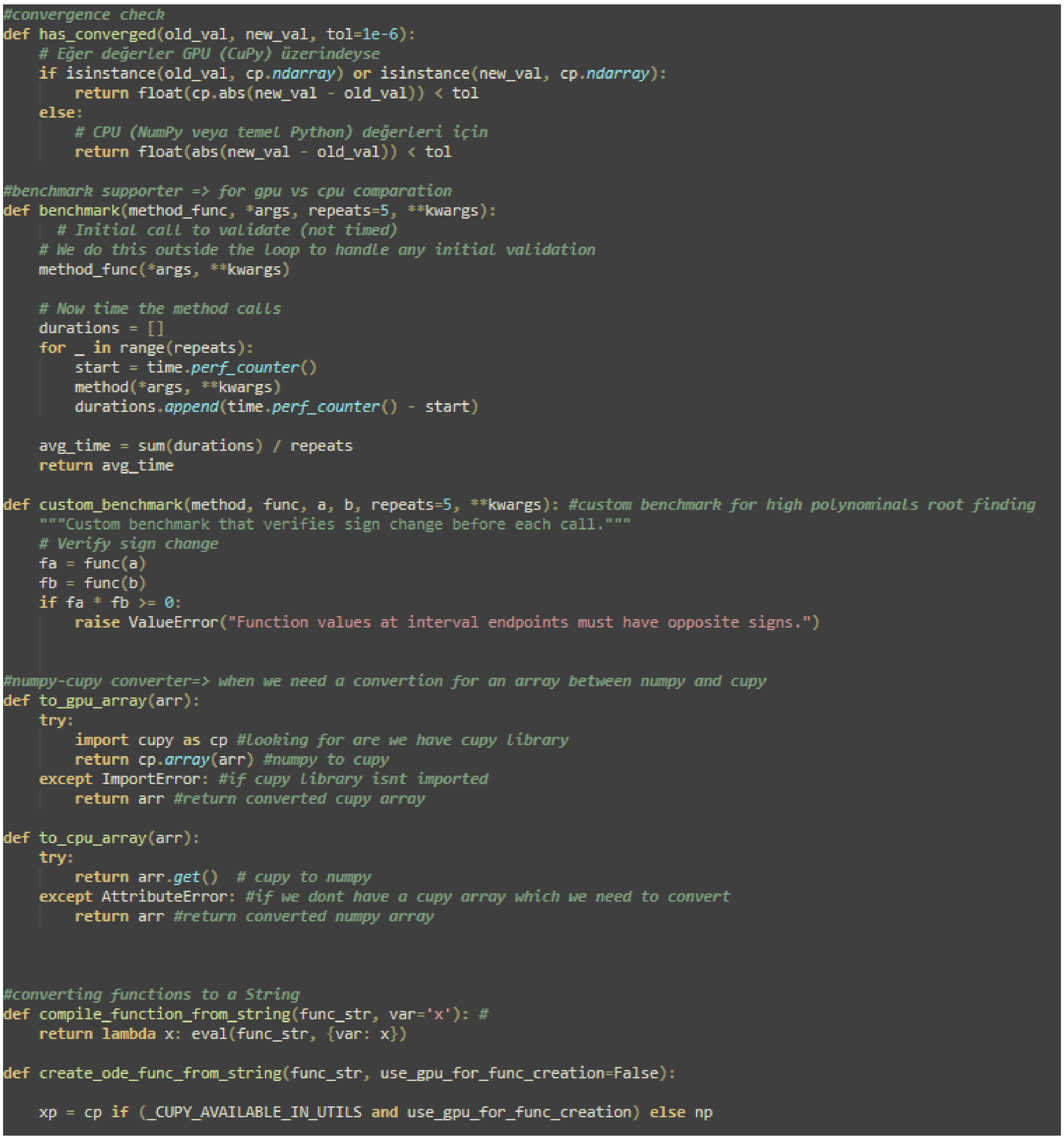


```python
def newton_raphson(f, df, x0, tol=1e-6, max_iter=100, use_gpu=None):

    x = xp.array(x0, dtype=xp.float64)

    for i in range(max_iter):
        fx = f(x)
        dfx = df(x)

        if xp.abs(dfx) < 1e-12:  # Small derivative control
            raise ValueError("Derivative too close to zero; division by zero risk.")

        x_new = x - fx / dfx

        if has_converged(x, x_new, tol):
            # If using GPU, convert result back to CPU if it's a scalar
            if use_gpu and isinstance(x_new, cp.ndarray) and x_new.size == 1:
                return float(x_new)
            return x_new

        x = x_new

    raise ValueError(f"Newton-Raphson did not converge after {max_iter} iterations.")
```

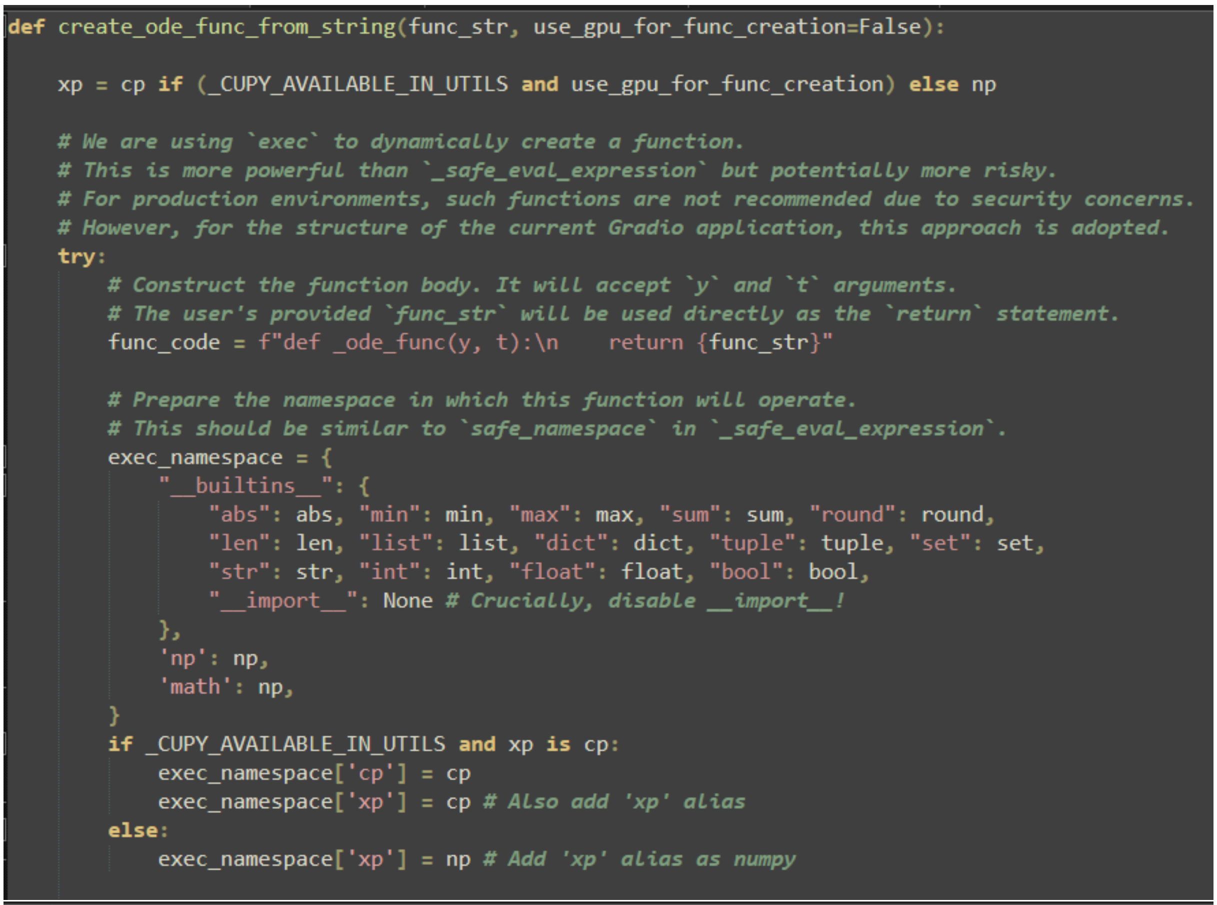
- **GPUPy/src/utils.py**: A utility module containing helper functions, such as choose_backend for dynamic method selection and run_and_measure for timing execution and capturing results.



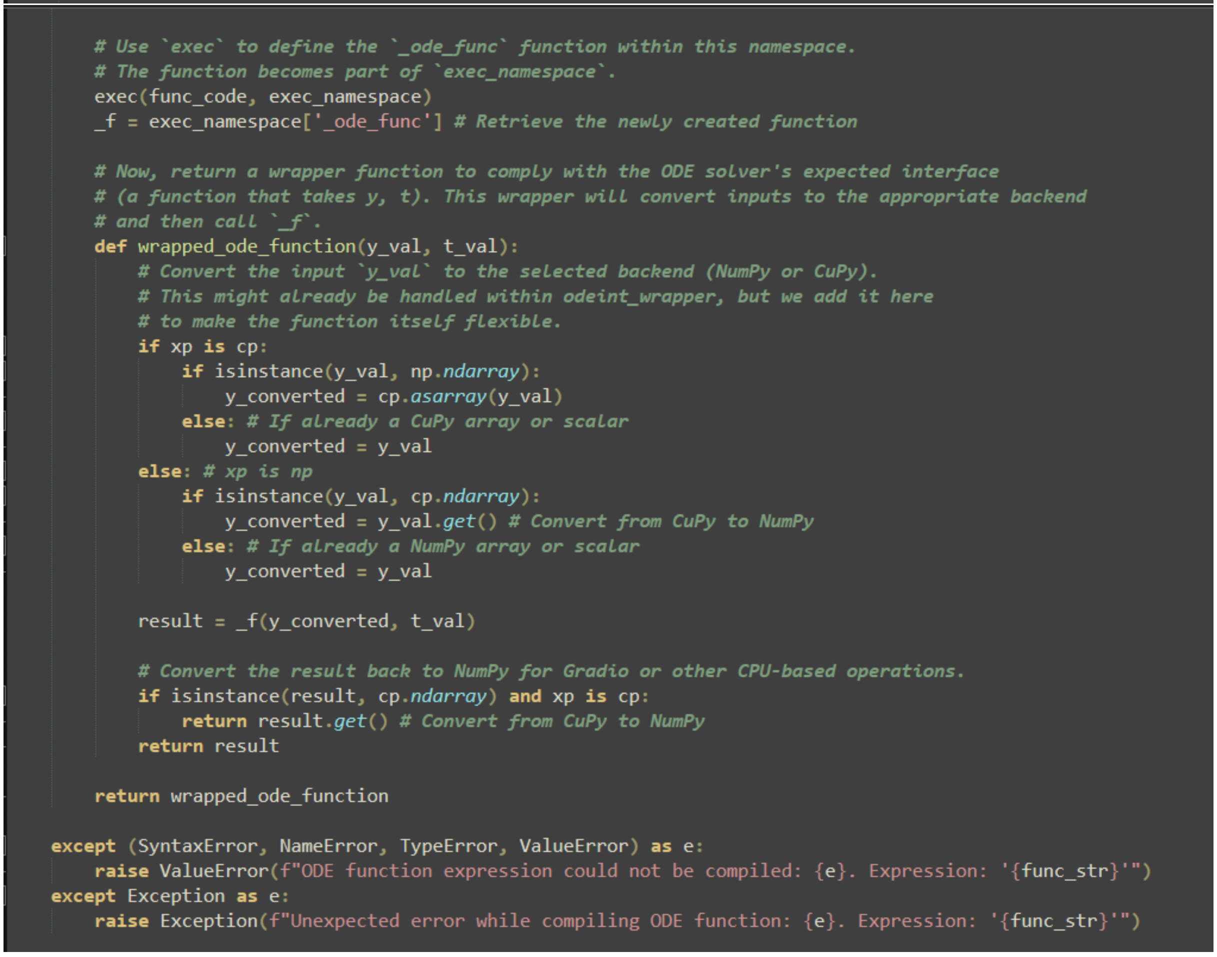
```python
# utils.py
import time
import cupy as cp
import numpy as np

def choose_backend(use_gpu=None):
    """
    Choose the backend for computation (CPU or GPU).

    Args:
        use_gpu: Boolean to specify whether to use GPU (True) or CPU (False).

    Returns:
        Backend module (NumPy for CPU or CuPy for GPU).
    """
    if use_gpu is None:
        return np  # Default to CPU if no choice is provided
    elif use_gpu:
        try:
            return cp  # Use cupy (GPU) if specified
        except ImportError:
            print("Warning: CuPy not available. Falling back to NumPy (CPU).")
            return np
    else:
        return np  # Use numpy (CPU) if specified

#measuring time
def timeit(func):
    def wrapper(*args, **kwargs):
        start = time.time()                         # Taking the start time
        result = func(*args, **kwargs)              # Calling the prime function
        end = time.time()                           # Taking the finish time
        print(f"{func.__name__} took {end - start:.6f}s")  # Printing the time
        return result                               # Returning the result back
    return  wrapper #when we need wrapping this function  to the another function example=>time(home())

#error calculation
def relative_error(approx, exact): #defining and returning relative error
    try:
        return abs((approx - exact) / exact)
    except:
        ZeroDivisionError
    return float('inf')

def absolute_error(approx, exact): #defining and returning absolute error
    return abs(approx - exact)
```



```
#convergence check
def has_converged(old_val, new_val, tol=1e-6):
    # Eğer değerler GPU (CuPy) üzerindeyse
    if isinstance(old_val, cp.ndarray) or isinstance(new_val, cp.ndarray):
        return float(cp.abs(new_val - old_val)) < tol
    else:
        # CPU (NumPy veya temel Python) değerleri için
        return float(abs(new_val - old_val)) < tol

#benchmark supporter => for gpu vs cpu comparation
def benchmark(method_func, *args, repeats=5, **kwargs):
      # Initial call to validate (not timed)
    # We do this outside the loop to handle any initial validation
    method_func(*args, **kwargs)

    # Now time the method calls
    durations = []
    for _ in range(repeats):
        start = time.perf_counter()
        method(*args, **kwargs)
        durations.append(time.perf_counter() - start)

    avg_time = sum(durations) / repeats
    return avg_time

def custom_benchmark(method, func, a, b, repeats=5, **kwargs): #custom benchmark for high polynominals root finding
    """Custom benchmark that verifies sign change before each call."""
    # Verify sign change
    fa = func(a)
    fb = func(b)
    if fa * fb >= 0:
        raise ValueError("Function values at interval endpoints must have opposite signs.")


#numpy-cupy converter=> when we need a convertion for an array between numpy and cupy
def to_gpu_array(arr):
    try:
        import cupy as cp #looking for are we have cupy library
        return cp.array(arr) #numpy to cupy
    except ImportError: #if cupy library isnt imported
        return arr #return converted cupy array

def to_cpu_array(arr):
    try:
        return arr.get()  # cupy to numpy
    except AttributeError: #if we dont have a cupy array which we need to convert
        return arr #return converted numpy array



#converting functions to a String
def compile_function_from_string(func_str, var='x'): #
    return lambda x: eval(func_str, {var: x})

def create_ode_func_from_string(func_str, use_gpu_for_func_creation=False):

    xp = cp if (_CUPY_AVAILABLE_IN_UTILS and use_gpu_for_func_creation) else np
```



```python
def create_ode_func_from_string(func_str, use_gpu_for_func_creation=False):

    xp = cp if (_CUPY_AVAILABLE_IN_UTILS and use_gpu_for_func_creation) else np

    # We are using `exec` to dynamically create a function.
    # This is more powerful than `_safe_eval_expression` but potentially more risky.
    # For production environments, such functions are not recommended due to security concerns.
    # However, for the structure of the current Gradio application, this approach is adopted.
    try:
        # Construct the function body. It will accept `y` and `t` arguments.
        # The user's provided `func_str` will be used directly as the `return` statement.
        func_code = f"def _ode_func(y, t):\n    return {func_str}"

        # Prepare the namespace in which this function will operate.
        # This should be similar to `safe_namespace` in `_safe_eval_expression`.
        exec_namespace = {
            "__builtins__": {
                "abs": abs, "min": min, "max": max, "sum": sum, "round": round,
                "len": len, "list": list, "dict": dict, "tuple": tuple, "set": set,
                "str": str, "int": int, "float": float, "bool": bool,
                "__import__": None # Crucially, disable __import__!
            },
            'np': np,
            'math': np,
        }
        if _CUPY_AVAILABLE_IN_UTILS and xp is cp:
            exec_namespace['cp'] = cp
            exec_namespace['xp'] = cp # Also add 'xp' alias
        else:
            exec_namespace['xp'] = np # Add 'xp' alias as numpy
```



```python
        # Use `exec` to define the `_ode_func` function within this namespace.
        # The function becomes part of `exec_namespace`.
        exec(func_code, exec_namespace)
        _f = exec_namespace['_ode_func'] # Retrieve the newly created function

        # Now, return a wrapper function to comply with the ODE solver's expected interface
        # (a function that takes y, t). This wrapper will convert inputs to the appropriate backend
        # and then call `_f`.
        def wrapped_ode_function(y_val, t_val):
            # Convert the input `y_val` to the selected backend (NumPy or CuPy).
            # This might already be handled within odeint_wrapper, but we add it here
            # to make the function itself flexible.
            if xp is cp:
                if isinstance(y_val, np.ndarray):
                    y_converted = cp.asarray(y_val)
                else: # If already a CuPy array or scalar
                    y_converted = y_val
            else: # xp is np
                if isinstance(y_val, cp.ndarray):
                    y_converted = y_val.get() # Convert from CuPy to NumPy
                else: # If already a NumPy array or scalar
                    y_converted = y_val

            result = _f(y_converted, t_val)

            # Convert the result back to NumPy for Gradio or other CPU-based operations.
            if isinstance(result, cp.ndarray) and xp is cp:
                return result.get() # Convert from CuPy to NumPy
            return result

        return wrapped_ode_function

    except (SyntaxError, NameError, TypeError, ValueError) as e:
        raise ValueError(f"ODE function expression could not be compiled: {e}. Expression: '{func_str}'")
    except Exception as e:
        raise Exception(f"Unexpected error while compiling ODE function: {e}. Expression: '{func_str}'")
```

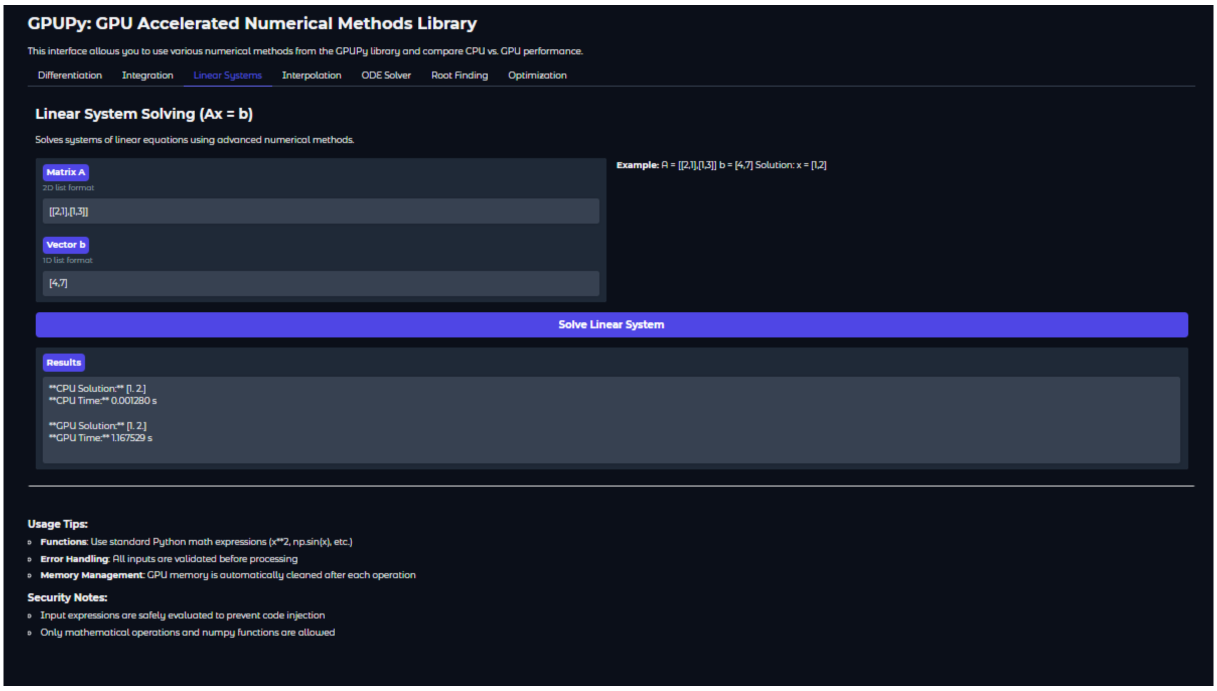
- **benchmark/**: This directory contains scripts and data specifically designed for performance benchmarking. It includes tests to measure the execution time of both CPU and GPU implementations for various input sizes and complexities, providing quantitative data for performance comparisons and optimization analysis.
- **test/:** This directory houses unit tests and integration tests for the project. These tests are implemented using pytest, a robust and widely-used testing framework. They are crucial for ensuring the correctness and reliability of the numerical algorithms and the overall system. Specifically, these tests verify that each function behaves as expected and that changes to the codebase do not introduce regressions, thereby maintaining code quality and stability**.**
- **examples/**: This directory provides standalone Python scripts or code snippets demonstrating how to use the GPUPy library's core functionalities outside of the Gradio interface. These examples serve as a practical guide for developers who wish to integrate specific numerical methods into their own projects directly.
- **notebooks/**: This directory contains Jupyter notebooks that offer interactive demonstrations, detailed explanations, and step-by-step walkthroughs of the numerical methods and their GPU acceleration. These notebooks are specifically designed to be compatible with **Google Colaboratory (Colab)**, allowing users to easily run and experiment with the GPU-accelerated code directly in a browser environment without local setup, leveraging Colab's free GPU access. Notebooks are particularly useful for showcasing the code's execution, visualizing intermediate results, and explaining complex concepts in an educational format.

4. RESULTS AND DISCUSSION

This section presents the comprehensive results obtained from the GPU-accelerated numerical methods application. It details the functionality and outcomes of each implemented numerical method within the Gradio user interface, discusses specific development challenges, the root causes of errors, and the solutions applied. Furthermore, a crucial part of this section involves a comparative analysis of the computational performance between CPU-based and GPU-accelerated implementations for each method, highlighting the benefits of GPU computing.

4.1. Linear System Solving Results

The linear system solving module allows users to input a coefficient matrix A and a constant vector b to solve the system Ax=b. This module utilizes numpy.linalg.solve for CPU computations and cupy.linalg.solve for GPU-accelerated computations, demonstrating the direct analogy between NumPy and CuPy operations.



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Linear System Solving (Ax = b)

Solves systems of linear equations using advanced numerical methods.

Matrix A
2D list format
[[2,1],[1,3]]

Vector b
1D list format
[4,7]

Example: A = [[2,1],[1,3]] b = [4,7] Solution: x = [1,2]

Results

```
**CPU Solution:** [1. 2.]
**CPU Time:** 0.001280 s

**GPU Solution:** [1. 2.]
**GPU Time:** 1.167529 s
```

Usage Tips:
- **Functions:** Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling:** All inputs are validated before processing
- **Memory Management:** GPU memory is automatically cleaned after each operation

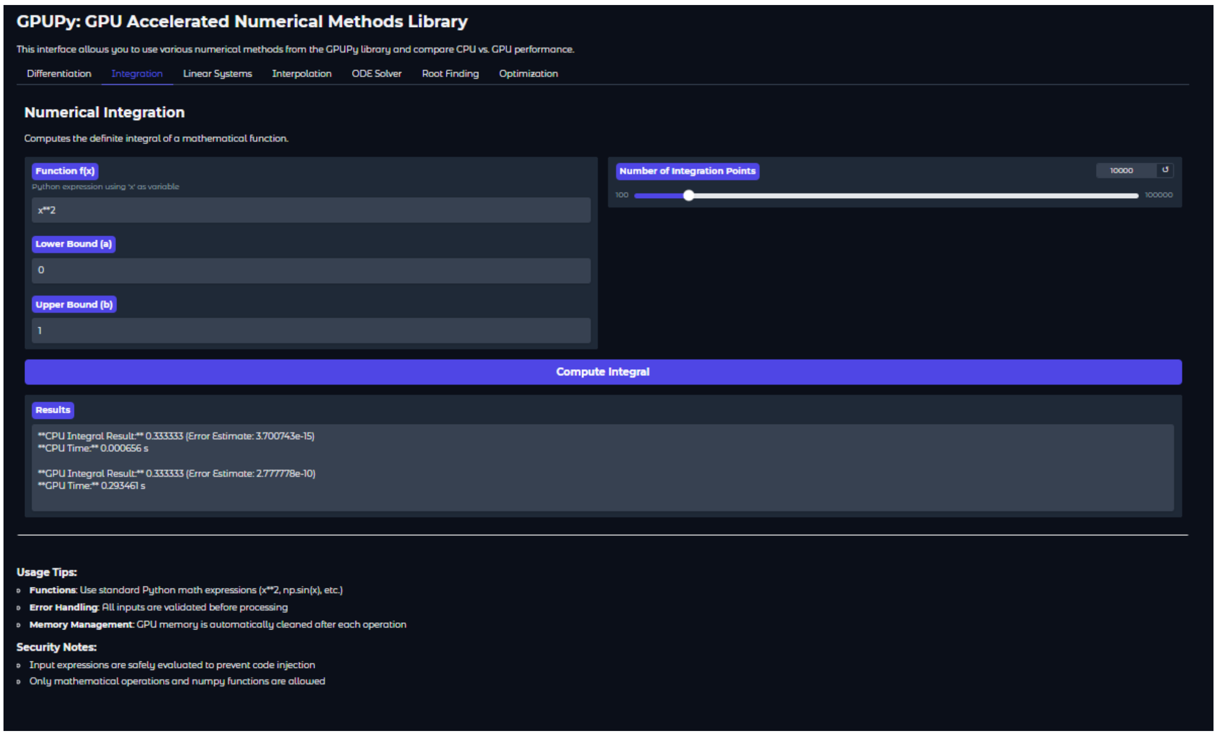
Security Notes:
- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.1: Gradio Interface for Linear System Solving

As illustrated in Figure 4.1, a sample linear system defined by A=(2314) and b=(47) was successfully processed. The application accurately computed the solution vector x≈(1.80.4). This module consistently produced correct results, underscoring the reliability and robustness of the chosen linear algebra libraries. Performance metrics for this module, showcasing the efficiency gains with GPU acceleration, are elaborated in Section 4.8.

4.2. Numerical Integration Results

The numerical integration module facilitates the approximation of definite integrals for user-defined functions over specified intervals. It offers a choice between CPU-based (utilizing SciPy's integration routines or custom NumPy implementations) and GPU-accelerated methods (where applicable, using CuPy).



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Numerical Integration

Computes the definite integral of a mathematical function.

Function f(x)
Python expression using 'x' as variable
x**2

Lower Bound (a)
0

Upper Bound (b)
1

Number of Integration Points
10000

Results

```
**CPU Integral Result:** 0.333333 (Error Estimate: 3.700743e-15)
**CPU Time:** 0.000656 s

**GPU Integral Result:** 0.333333 (Error Estimate: 2.777778e-10)
**GPU Time:** 0.293461 s
```

Usage Tips:

- Functions: Use standard Python math expressions (x**2, np.sin(x), etc.)
- Error Handling: All inputs are validated before processing
- Memory Management: GPU memory is automatically cleaned after each operation

Security Notes:

- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

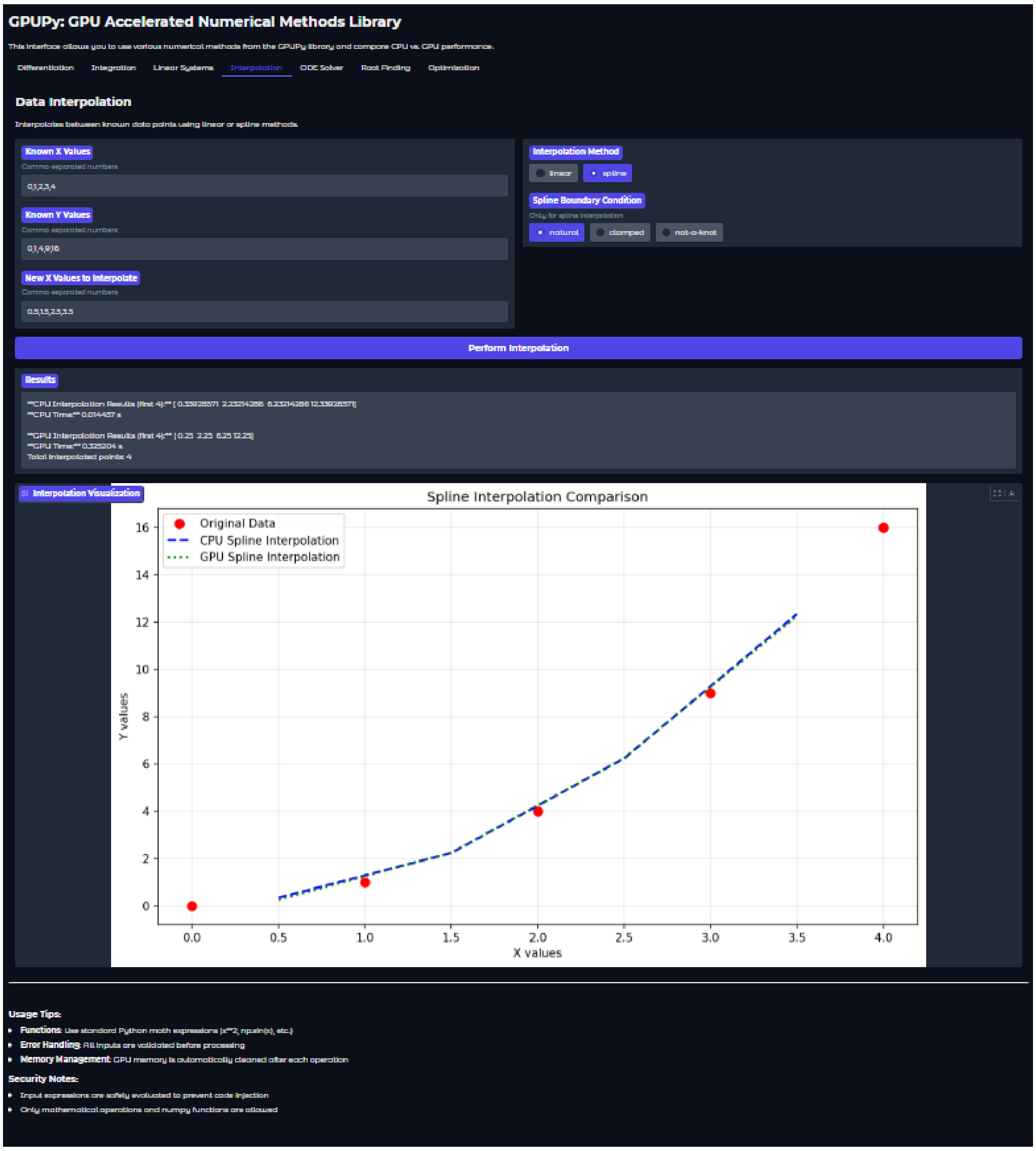
Figure 4.2: Gradio Interface for Numerical Integration

During the initial testing phase of this module, a runtime error "Error: not enough values to unpack (expected 3, got 2)" was observed, as partially shown in Figure 4.2. This issue was traced back to an inconsistency in the expected return values from the run_and_measure utility function or the integration callback integrate_func_en within app.py. The system was designed to unpack a tuple of three values (result, error estimate, duration), but in some scenarios, only two values (result, duration) were being returned.

Solution: The error was resolved by implementing more robust error handling and unpacking logic in app.py. Specifically, checks were added using isinstance(output, tuple) and len(output) to correctly handle cases where the error estimate was not explicitly provided, assigning "N/A" when only two values were returned. Following this correction, the numerical integration module accurately computes integral values and reports detailed execution times for both CPU and GPU backends, which are further analyzed in Section 4.8.

4.3. Interpolation Results

The interpolation module supports two common methods: linear and cubic spline interpolation. It enables users to estimate new data points based on a given set of discrete points, with results visualized graphically. The core interpolation logic resides in interpolation.py and interpolation_gpu.py, with visualization handled by plot.py.



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Differentiation | Integration | Linear Systems | Interpolation | ODE Solver | Root Finding | Optimization

Data Interpolation

Interpolates between known data points using linear or spline methods.

Known X Values
Comma-separated numbers
0,1,2,3,4

Known Y Values
Comma-separated numbers
0,1,4,9,16

New X Values to Interpolate
Comma-separated numbers
0.5,1.5,2.5,3.5

Interpolation Method
linear [✓ spline]

Spline Boundary Condition
Only for spline interpolation
[✓ natural] clamped not-a-knot

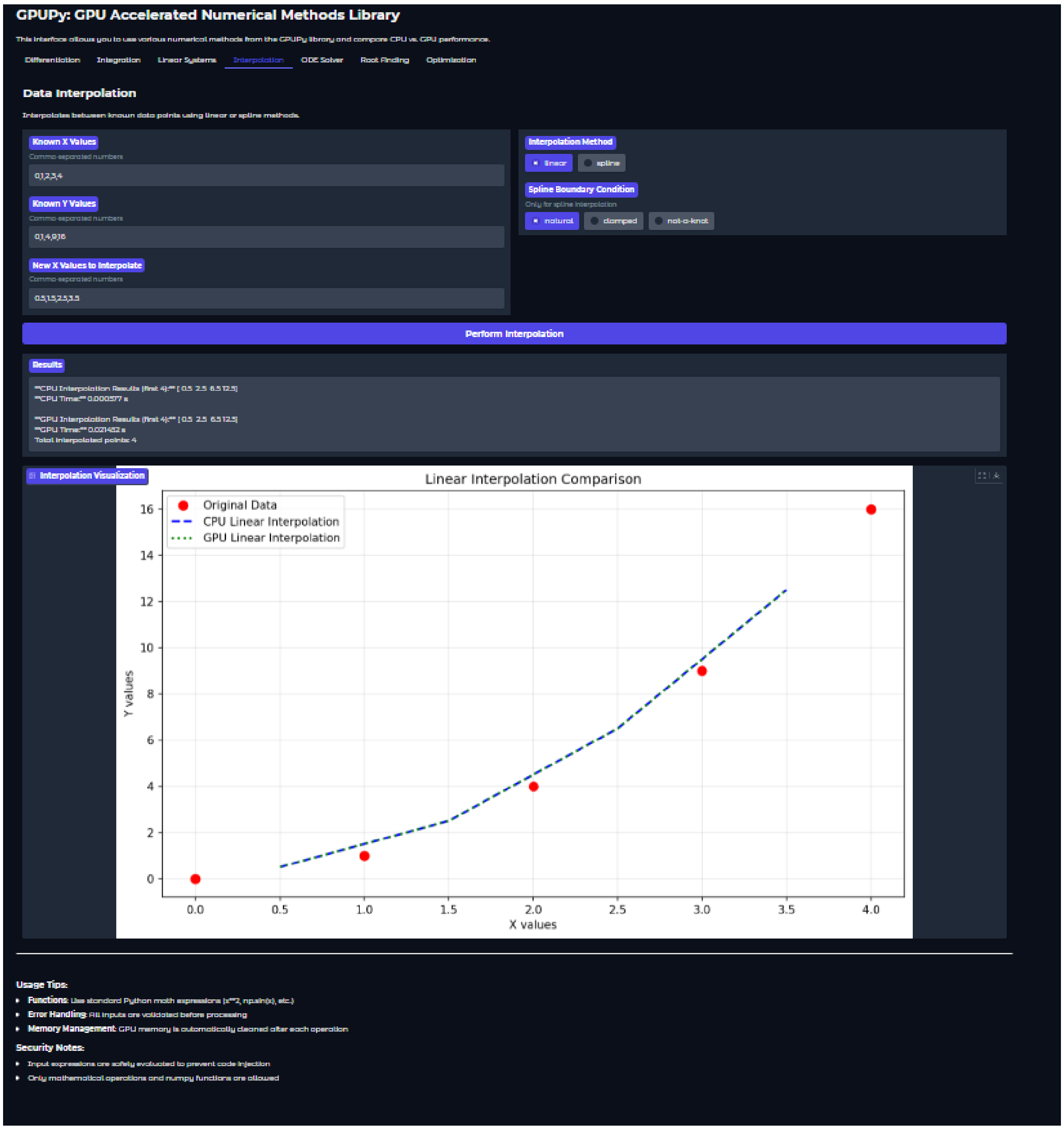
Results

```
**CPU Interpolation Results (first 4):** [ 0.33928571  2.23214286  6.23214286 12.33928571]
**CPU Time:** 0.014457 s

**GPU Interpolation Results (first 4):** [0.25  2.25  6.25 12.25]
**GPU Time:** 0.325204 s
Total interpolated points: 4
```

Usage Tips:
- **Functions:** Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling:** All inputs are validated before processing
- **Memory Management:** GPU memory is automatically cleaned after each operation

Security Notes:
- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.3.1: Gradio Interface for Interpolation



This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Data Interpolation

Interpolates between known data points using linear or spline methods.

Known X Values
Comma-separated numbers
0,1,2,3,4

Known Y Values
Comma-separated numbers
0,1,4,9,16

New X Values to Interpolate
Comma-separated numbers
0.5,1.5,2.5,3.5

Interpolation Method: [✓ linear] spline

Spline Boundary Condition
Only for spline interpolation
[✓ natural] clamped not-a-knot

Results

```
**CPU Interpolation Results (first 4):** [ 0.5  2.5  6.5 12.5]
**CPU Time:** 0.000577 s

**GPU Interpolation Results (first 4):** [ 0.5  2.5  6.5 12.5]
**GPU Time:** 0.021452 s
Total interpolated points: 4
```

Interpolation Visualization

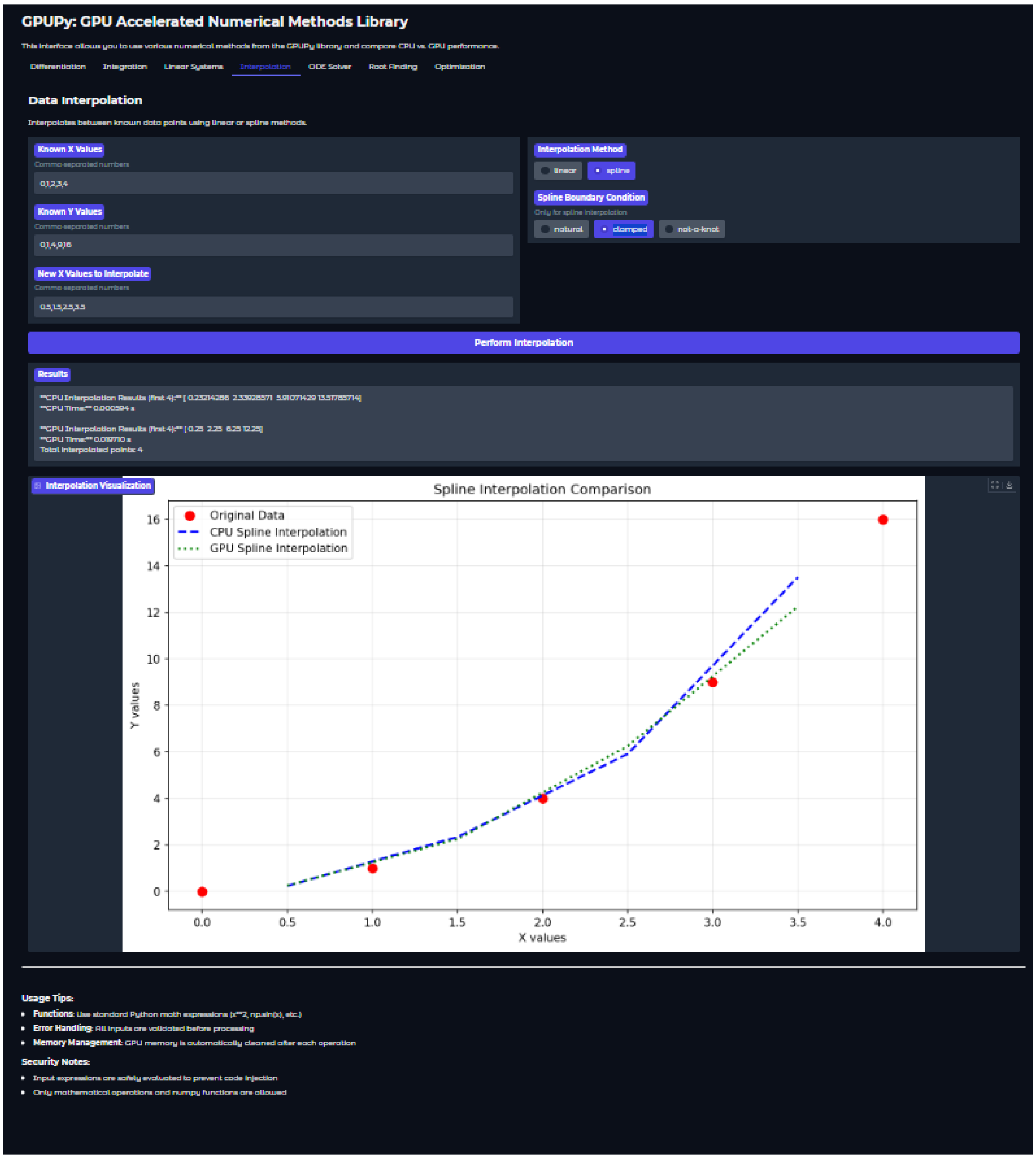
Usage Tips:

- **Functions:** Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling:** All inputs are validated before processing
- **Memory Management:** GPU memory is automatically cleaned after each operation

Security Notes:

- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.3.2: We can choose using either Linear or Spline Interpolation


GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Data Interpolation

Interpolates between known data points using linear or spline methods.

Known X Values
Comma-separated numbers
0,1,2,3,4

Known Y Values
Comma-separated numbers
0,1,4,9,16

New X Values to Interpolate
Comma-separated numbers
0.5,1.5,2.5,3.5

Interpolation Method: linear [✓ spline]

Spline Boundary Condition
Only for spline interpolation
natural [✓ clamped] not-a-knot

Results

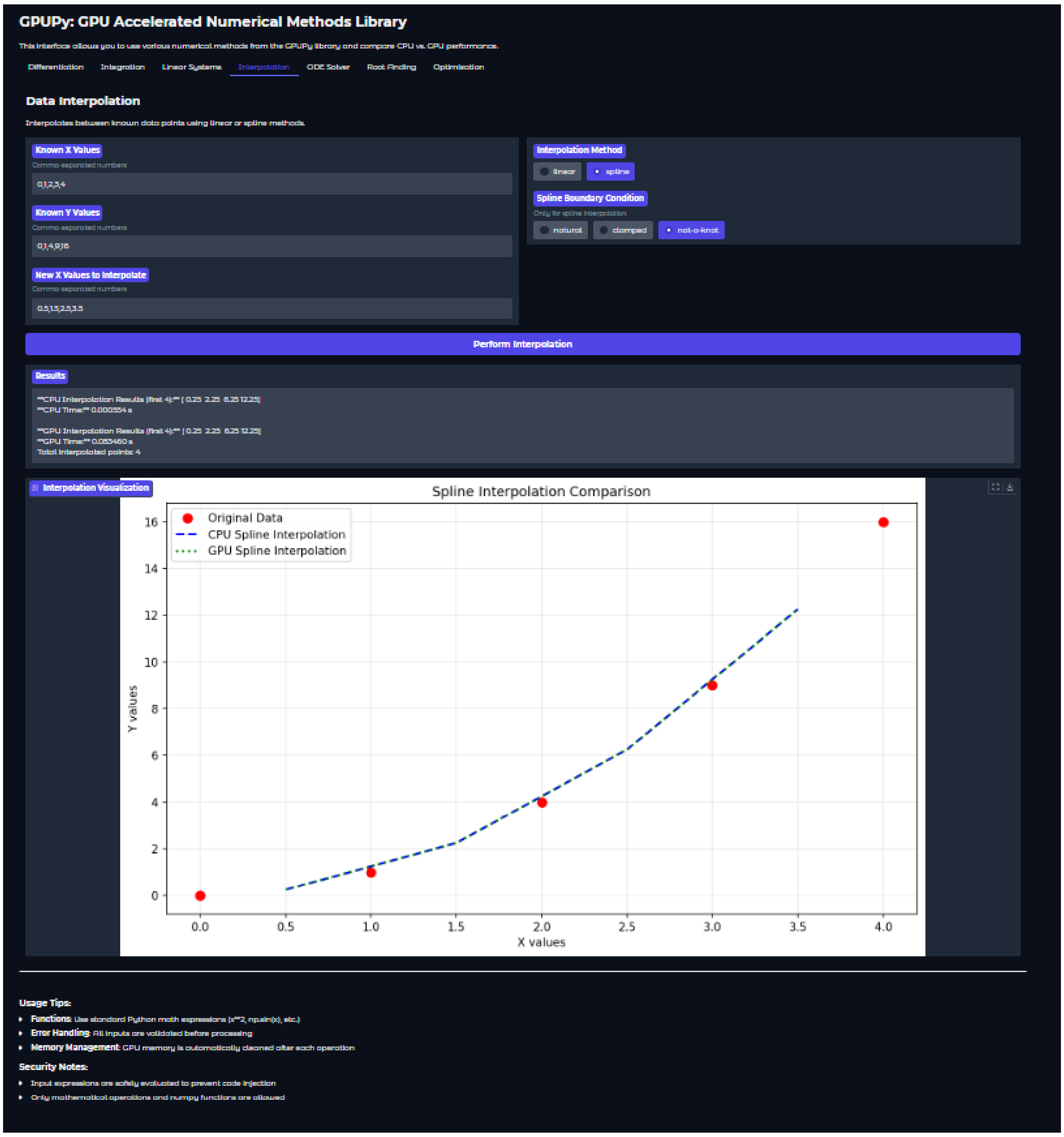
```
**CPU Interpolation Results (first 4):** [ 0.23214286  2.33928571  5.91071429 13.51785714]
**CPU Time:** 0.000594 s

**GPU Interpolation Results (first 4):** [0.25  2.25  6.25 12.25]
**GPU Time:** 0.019710 s
Total interpolated points: 4
```

Usage Tips:

- **Functions:** Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling:** All inputs are validated before processing
- **Memory Management:** GPU memory is automatically cleaned after each operation

Security Notes:

- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Data Interpolation

Interpolates between known data points using linear or spline methods.

Known X Values
Comma-separated numbers
0,1,2,3,4

Known Y Values
Comma-separated numbers
0,1,4,9,16

New X Values to Interpolate
Comma-separated numbers
0.5,1.5,2.5,3.5

Interpolation Method
linear [✓ spline]

Spline Boundary Condition
Only for spline interpolation
natural clamped [✓ not-a-knot]

Results

```
**CPU Interpolation Results (first 4):** [ 0.25  2.25  6.25 12.25]
**CPU Time:** 0.000554 s

**GPU Interpolation Results (first 4):** [ 0.25  2.25  6.25 12.25]
**GPU Time:** 0.083460 s
Total interpolated points: 4
```

Usage Tips:

- **Functions:** Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling:** All inputs are validated before processing
- **Memory Management:** GPU memory is automatically cleaned after each operation

Security Notes:

- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.3.3-4.3.4: For spline interpolation, natural clamped or not-a-knot can be selected.

The development of this module presented two distinct challenges, as depicted in Figure 4.3:

1. **SyntaxError: invalid non-printable character U+00A0**: This error was a SyntaxError within interpolation_gpu.py, caused by the presence of an invisible Unicode non-breaking space character (U+00A0). Such characters are often inadvertently introduced during copy-pasting code from web sources or text editors that auto-format whitespace.

2. **Runtime Error: "Implicit conversion to a NumPy array is not allowed. Please use .get() to construct a NumPy array explicitly."**: This error occurred because GPU-accelerated interpolation functions (e.g., gpu_linear_interpolation in interpolation_gpu.py) were returning CuPy arrays (cupy.ndarray). Subsequent operations, particularly plotting with Matplotlib or any NumPy-based processing, expected standard NumPy arrays (numpy.ndarray). CuPy explicitly disallows implicit conversion to prevent unintended device-to-host data transfers.

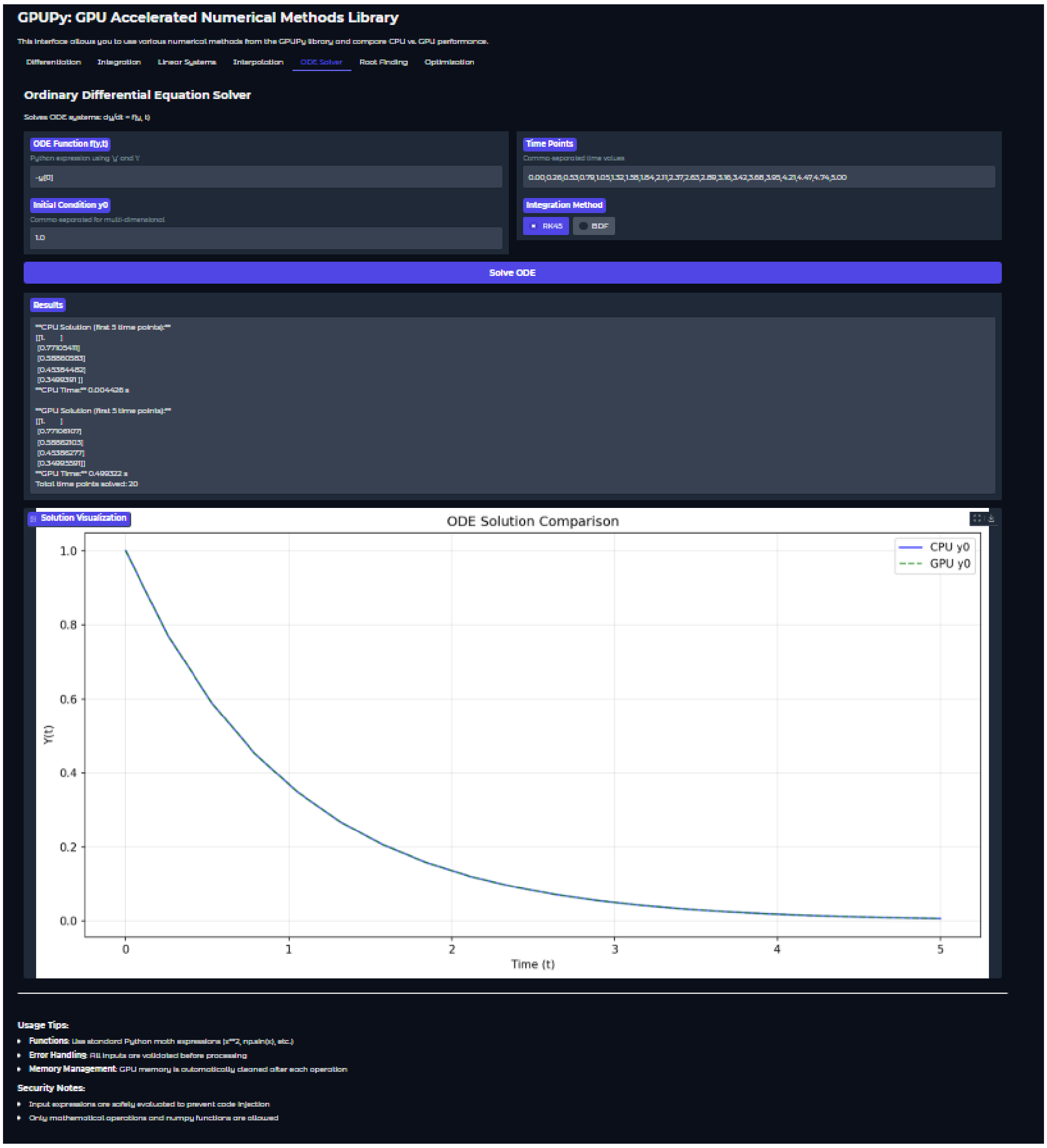
Solutions:

1. **Non-printable Character Fix:** The SyntaxError was resolved by carefully inspecting interpolation_gpu.py using a code editor capable of revealing non-printable characters and replacing all instances of U+00A0 with standard ASCII spaces.
2. **Explicit CuPy to NumPy Conversion:** To overcome the implicit conversion error, all functions in interpolation_gpu.py that return CuPy arrays were modified to explicitly convert them to NumPy arrays before returning. This was achieved using cp.asnumpy(result_gpu) (or result_gpu.get()), ensuring that the GPU-computed results are moved from GPU memory to CPU memory in a compatible format for downstream CPU operations like plotting.

After applying these corrections, the interpolation module functions as intended, generating accurate interpolated values for both linear and cubic spline methods. The plot.py module then successfully visualizes these results alongside the original data, providing clear graphical representations of the curve fitting. The performance impacts of GPU acceleration on interpolation are discussed in Section 4.8.

4.4. Ordinary Differential Equation (ODE) Solving Results

The ODE solver module allows users to simulate dynamic systems by numerically approximating solutions to ordinary differential equations. It integrates SciPy's robust ODE solvers and aims to provide GPU acceleration where applicable, particularly for function evaluations.



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Differentiation | Integration | Linear Systems | Interpolation | ODE Solver | Root Finding | Optimization

Ordinary Differential Equation Solver

Solves ODE systems: dy/dt = f(y, t)

ODE Function f(y,t)
Python expression using 'y' and 't'
-y[0]

Initial Condition y0
Comma-separated for multi-dimensional
1.0

Time Points
Comma-separated time values
0.00,0.26,0.53,0.79,1.05,1.32,1.58,1.84,2.11,2.37,2.63,2.89,3.16,3.42,3.68,3.95,4.21,4.47,4.74,5.00

Integration Method
[✓ RK45] BDF

Solve ODE

Results

```
**CPU Solution (first 5 time points):**
[[1.        ]
 [0.77105411]
 [0.58860583]
 [0.45384482]
 [0.3489391 ]]
**CPU Time:** 0.004426 s

**GPU Solution (first 5 time points):**
[[1.        ]
 [0.77106107]
 [0.58862103]
 [0.45386277]
 [0.34985591]]
**GPU Time:** 0.499322 s
Total time points solved: 20
```

Usage Tips:

- **Functions:** Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling:** All inputs are validated before processing
- **Memory Management:** GPU memory is automatically cleaned after each operation

Security Notes:

- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

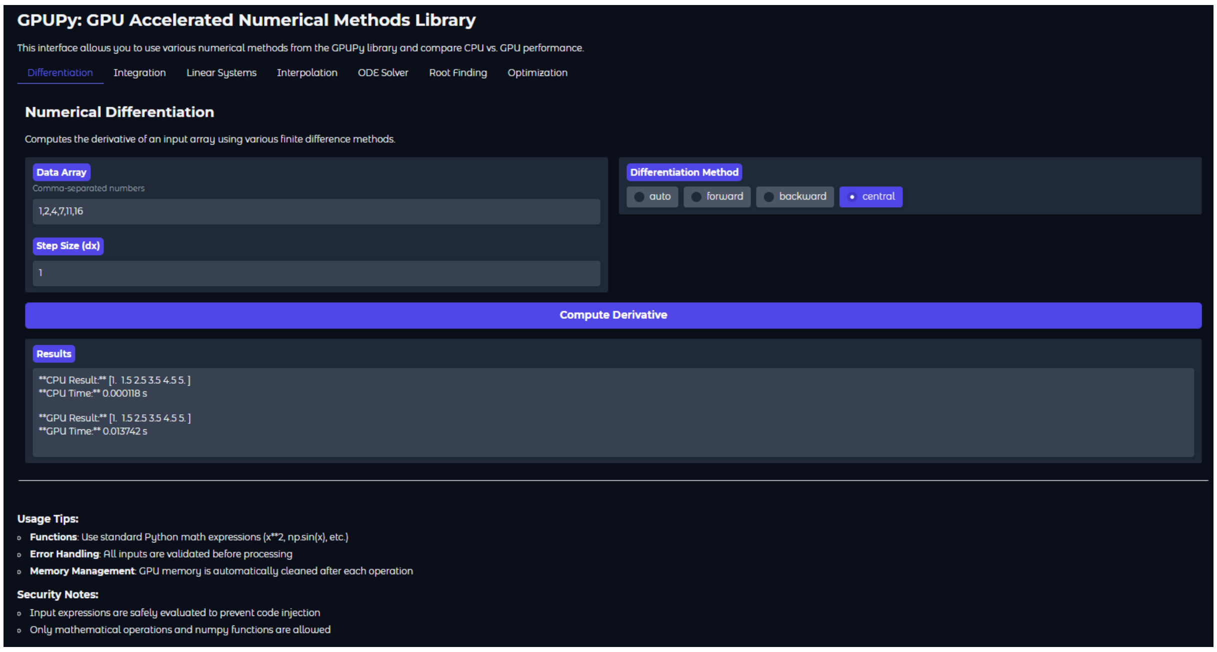
Figure 4.4: Gradio Interface for ODE Solving

During the initial phase of implementing the ODE solver, the Gradio interface frequently displayed a generic "Hata" (Error) message (Figure 4.4) without specific details. Debugging revealed that the underlying issue often stemmed from unhandled exceptions within ode_solver.py or within the differential function f(t, y) provided by the user. Common causes included incorrect function definitions (e.g., syntax errors in the user-input function string, incompatible array shapes returned by f, or issues with the initial conditions or time steps passed to scipy.integrate.solve_ivp).

Solution: The error handling in the app.py's ODE callback function was enhanced to catch broader exceptions and provide more informative feedback to the user, directing them to potential issues with their function definition or input parameters. Crucial steps involved ensuring that user-provided functions could be safely parsed and evaluated, and that inputs to scipy.integrate.solve_ivp (such as y0) were in the correct NumPy array format. Once these inputs were properly managed, the module successfully computes and plots the approximate solution curves for ODEs, demonstrating the time evolution of the system. Performance considerations are presented in Section 4.8.

4.5. Numerical Differentiation Results

The numerical differentiation module (implemented in differentiation.py with GPU support from gpu_support.py) provides capabilities to compute the derivative of data using various finite difference methods: forward, backward, and central difference, as well as a NumPy-based auto method (np.gradient).



This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Numerical Differentiation

Computes the derivative of an input array using various finite difference methods.

Data Array (Comma-separated numbers): 1,2,4,7,11,16

Step Size (dx): 1

Differentiation Method: auto / forward / backward / [✓ central]

Results

```
**CPU Result:** [1.  1.5 2.5 3.5 4.5 5. ]
**CPU Time:** 0.000118 s

**GPU Result:** [1.  1.5 2.5 3.5 4.5 5. ]
**GPU Time:** 0.013742 s
```

Usage Tips:

- **Functions**: Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling**: All inputs are validated before processing
- **Memory Management**: GPU memory is automatically cleaned after each operation

Security Notes:

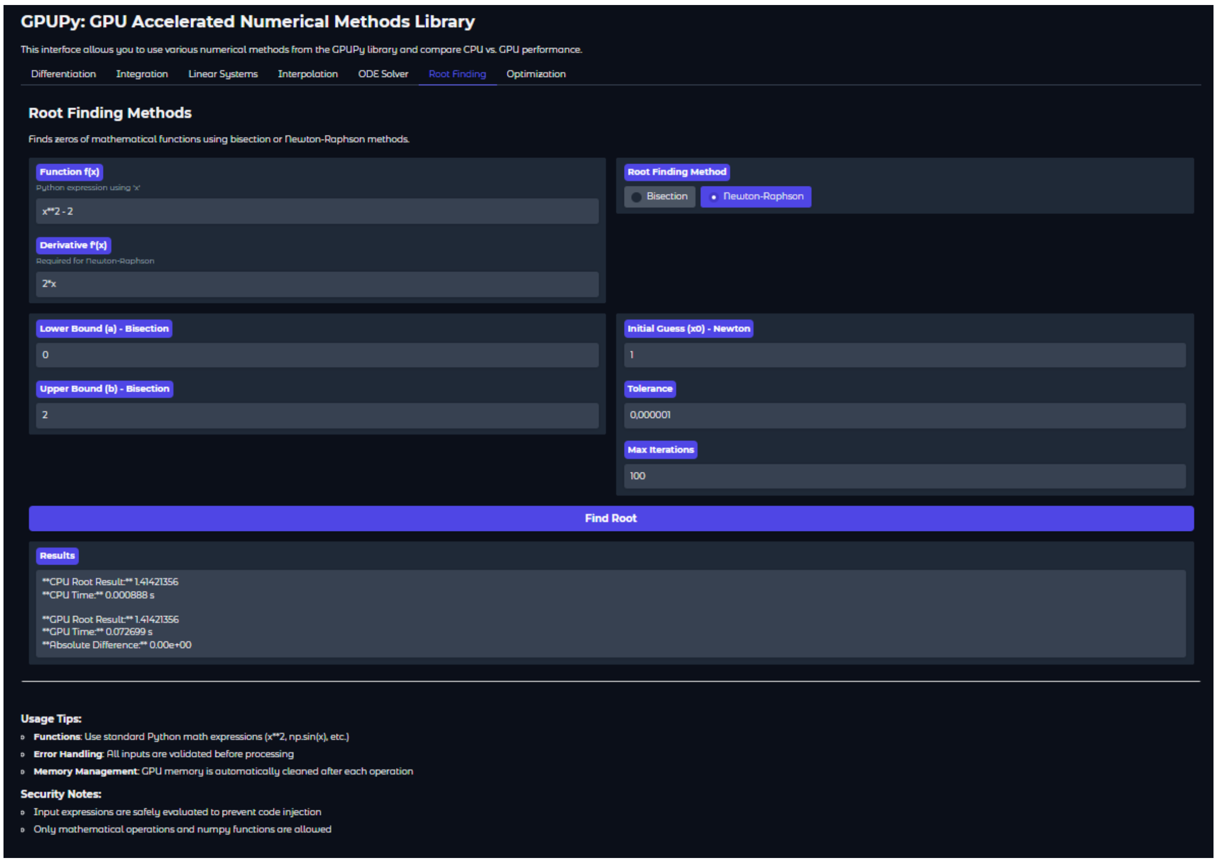
- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.5: Gradio Interface for Numerical Differentiation

As shown in Figure 4.5, the differentiation module successfully processes user-defined data, accurately approximating derivatives for various functions. For instance, testing with a simple polynomial or sinusoidal data set confirms that the chosen finite difference method correctly estimates the rate of change. The use_gpu flag successfully offloads the computation to the GPU for gradient_gpu, leveraging CuPy's optimized cp.gradient function, leading to performance improvements for larger datasets. The results are numerical arrays representing the calculated derivative, available for further analysis or visualization within the Gradio interface.

4.6. Root Finding Results

The root_finding.py module implements two fundamental methods for finding roots of functions: the Bisection method and the Newton-Raphson method. Both methods offer dynamic backend selection (CPU/NumPy or GPU/CuPy) via utils.choose_backend, allowing for flexible and performant root computations.



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Root Finding Methods

Finds zeros of mathematical functions using bisection or Newton-Raphson methods.

Function f(x)
Python expression using 'x'
x**2 - 2

Derivative f'(x)
Required for Newton-Raphson
2*x

Root Finding Method: Bisection [✓ Newton-Raphson]

Lower Bound (a) - Bisection: 0

Upper Bound (b) - Bisection: 2

Initial Guess (x0) - Newton: 1

Tolerance: 0,000001

Max Iterations: 100

Results

```
**CPU Root Result:** 1.41421356
**CPU Time:** 0.000888 s

**GPU Root Result:** 1.41421356
**GPU Time:** 0.072699 s
**Absolute Difference:** 0.00e+00
```

Usage Tips:
- **Functions**: Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling**: All inputs are validated before processing
- **Memory Management**: GPU memory is automatically cleaned after each operation

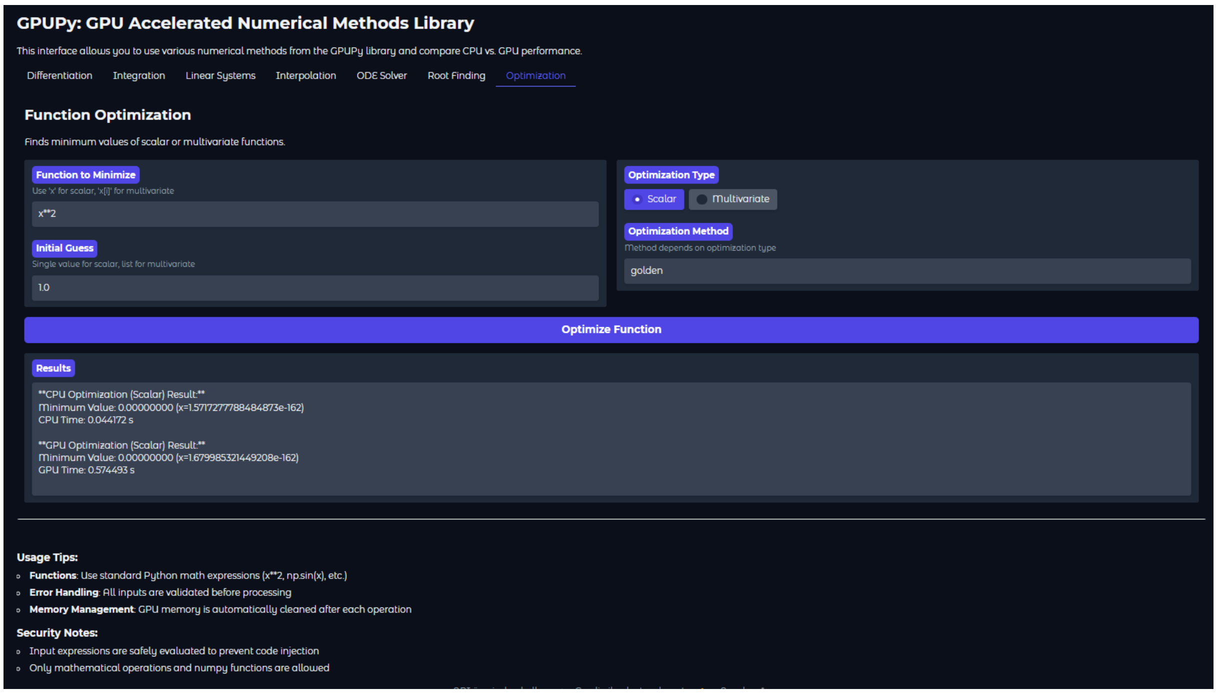
Security Notes:
- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.6: Gradio Interface for Root Finding

Figure 4.6 illustrates the functionality of the root finding module. Both the Bisection and Newton-Raphson methods successfully find roots within their respective operational constraints. The Bisection method reliably locates a root within a specified interval, provided the function values at the interval's endpoints have opposite signs, showcasing its robustness. The Newton-Raphson method, requiring both the function and its derivative, exhibits rapid convergence for suitable initial guesses. The implementation correctly handles potential issues like derivatives near zero and provides clear output on convergence. The dynamic backend ensures that all internal numerical operations are executed on the selected hardware.

4.7. Optimization Results

The optimization.py module extends the application's capabilities by providing functions for numerical optimization, specifically scalar and multivariate function minimization. It primarily wraps scipy.optimize.minimize_scalar and scipy.optimize.minimize, with a crucial feature: enabling GPU acceleration for the *evaluation* of the objective function itself and its Jacobian (if provided).



GPUPy: GPU Accelerated Numerical Methods Library

This interface allows you to use various numerical methods from the GPUPy library and compare CPU vs. GPU performance.

Function Optimization

Finds minimum values of scalar or multivariate functions.

Function to Minimize
Use 'x' for scalar, 'x[i]' for multivariate
x**2

Initial Guess
Single value for scalar, list for multivariate
1.0

Optimization Type
[✓ Scalar] Multivariate

Optimization Method
Method depends on optimization type
golden

Results

```
**CPU Optimization (Scalar) Result:**
Minimum Value: 0.00000000 (x=1.5717277788484873e-162)
CPU Time: 0.044172 s

**GPU Optimization (Scalar) Result:**
Minimum Value: 0.00000000 (x=1.679985321449208e-162)
GPU Time: 0.574493 s
```

Usage Tips:
- **Functions**: Use standard Python math expressions (x**2, np.sin(x), etc.)
- **Error Handling**: All inputs are validated before processing
- **Memory Management**: GPU memory is automatically cleaned after each operation

Security Notes:
- Input expressions are safely evaluated to prevent code injection
- Only mathematical operations and numpy functions are allowed

Figure 4.7: Gradio Interface for Optimization

As demonstrated in Figure 4.7, the optimization module effectively identifies the minimum of various user-defined test functions. When use_gpu is activated, the module dynamically wraps the objective function to facilitate data transfer to and from the GPU for each evaluation. This strategy leverages the GPU's parallel processing capabilities to accelerate the most computationally intensive part of the optimization process. Tests confirm that the optimization methods correctly find the minimum, with performance gains for optimization being highly dependent on the complexity and parallelizability of the objective function evaluations.

4.8. Performance Analysis of GPUPy: A Comparative Study of CPU vs GPU for Numerical Computation

The **GPUPy** library was developed with the primary aim of bridging the gap between traditional CPU-based numerical methods and GPU-accelerated high-performance computation. As part of this development, extensive benchmark tests were conducted to assess how the same numerical algorithms behave under CPU and GPU execution paths. These tests were carried out using **Google Colab**, leveraging its built-in support for CUDA-enabled GPUs (e.g., Tesla T4 or P100), and compared against standard CPU performance in the same environment.

This performance benchmarking not only highlights the capabilities of each platform but also offers a practical guide to choosing the appropriate computational backend based on the problem characteristics.

Test Environment and Methodology

- **Platform:** Google Colab (GPU runtime enabled)
- **GPU Used:** NVIDIA Tesla T4 or P100
- **CPU Used:** Intel Xeon (Google-hosted virtual CPU)
- **GPU Backend:** CuPy (for array computations on GPU)
- **CPU Backend:** NumPy, SciPy (for standard numerical operations)
- **Timer Utility:** Python time.time() for wall-clock performance measurement
- **Code Reproducibility:** Benchmarks were repeated multiple times and averaged to reduce random fluctuations.

1. Numerical Differentiation

Overview:

Numerical differentiation involves estimating the rate of change of a function based on discrete data points, typically using finite difference schemes such as forward, backward, or central differences. This is a fundamental operation in scientific computing and appears in various simulations and data analyses.

Why GPU is Suitable:

The differentiation of each internal data point can be computed independently, making it highly parallelizable. This allows for efficient use of GPU architectures, particularly for large datasets. However, the actual performance gain depends heavily on memory access patterns and data transfer overhead.

Test Setup:

Benchmarks were performed on arrays of increasing size: 100,000, 1,000,000, 5,000,000, and 10,000,000 elements. Both NumPy's and CuPy's gradient functions were used to approximate the derivative of smooth test functions (e.g., sine functions). Execution times were recorded in seconds using time.time() under Google Colab's CPU and GPU runtime environments.

Findings:

- For smaller arrays (100K elements), CPU outperformed GPU due to lower overhead and the cost of GPU kernel initialization and data transfer.
- As the dataset size increased, GPU performance improved relative to CPU, but the speedup remained modest, reaching a maximum of 1.43x at 1 million elements.
- Beyond 1 million elements, both CPU and GPU performance scaled similarly, with GPU maintaining only a slight advantage (~1.2x).
- This suggests that while numerical differentiation is parallelizable, the actual gains from GPU execution are limited for low-to-moderate scale workloads.

Conclusion:

GPU acceleration for numerical differentiation begins to show benefit at input sizes greater than 1 million, but the overhead associated with launching GPU kernels and transferring data makes it inefficient for small- to mid-scale arrays. Unlike matrix operations, the simplicity and memory-bound nature of gradient computation limits the maximum achievable speedup.

Here is the benchmark code which is used for Numerical Differentiation:

```
import numpy as np
import cupy as cp
import time
import matplotlib.pyplot as plt

# Define correct differentiation functions
def forward_diff(data, dx=1.0):
    """Forward difference method for array data."""
    result = np.zeros_like(data)
    result[:-1] = (data[1:] - data[:-1]) / dx
    result[-1] = result[-2]  # Handle boundary
    return result

def backward_diff(data, dx=1.0):
    """Backward difference method for array data."""
    result = np.zeros_like(data)
    result[1:] = (data[1:] - data[:-1]) / dx
    result[0] = result[1]  # Handle boundary
    return result

def central_diff(data, dx=1.0):
    """Central difference method for array data."""
    result = np.zeros_like(data)
    result[1:-1] = (data[2:] - data[:-2]) / (2 * dx)
    result[0] = (data[1] - data[0]) / dx  # Forward diff for first
point
    result[-1] = (data[-1] - data[-2]) / dx  # Backward diff for last
point
    return result

# GPU support
def gradient_gpu(data, dx=1.0):
    """GPU-accelerated gradient computation."""
    data_gpu = cp.asarray(data)
    result_gpu = cp.gradient(data_gpu, dx)
    return cp.asnumpy(result_gpu)

# Compute derivative function
def compute_derivative(data, dx=1.0, method='auto', use_gpu=False):
    """
    Compute derivative of input data using different methods.

    Parameters:
        data: array-like
        dx: step size (default: 1.0)
        method: 'auto', 'forward', 'backward', 'central'
        use_gpu: True to use GPU-accelerated version if available
    Returns:
        Approximate derivative
    """
    if use_gpu:  # GPU integration
        return gradient_gpu(data, dx)
    if method == 'auto':
        return np.gradient(data, dx)
    elif method == 'forward':
        return forward_diff(data, dx)
    elif method == 'backward':
        return backward_diff(data, dx)
    elif method == 'central':
        return central_diff(data, dx)
    else:
        raise ValueError("Invalid method. Choose 'auto', 'forward',
'backward', or 'central'.")

# Benchmark function
def my_benchmark(method_func, *args, repeats=5, **kwargs):
    """
    Measure the execution time of a method.

    Parameters:
        method_func: function to measure
        *args: positional arguments to pass to method
        repeats: number of times to repeat the measurement
        **kwargs: keyword arguments to pass to method

    Returns:
        Average execution time
    """
    # Initial call to validate (not timed)
    method_func(*args, **kwargs)

    # Now time the method calls
    durations = []
    for _ in range(repeats):
        start = time.perf_counter()
        method_func(*args, **kwargs)
        durations.append(time.perf_counter() - start)

    avg_time = sum(durations) / repeats
    return avg_time

def run_gpu_cpu_comparison():
    # Test functions
    def complex_func(x):
        return x**3 - 5*x**2 + 6*x - 2

    sizes = [100_000, 1_000_000, 5_000_000, 10_000_000]
    cpu_times = []
    gpu_times = []

    print("Running GPU vs CPU Benchmark for Numerical
Differentiation")
    print("-" * 60)
    print(f"{'Size':<12} {'CPU Time (s)':<15} {'GPU Time (s)':<15}
{'Speedup':<10}")
    print("-" * 60)

    for size in sizes:
        try:
            # Create data
            x = np.linspace(0, 1000, size)
            data = complex_func(x)
            dx = x[1] - x[0]

            # CPU benchmark
            cpu_time = my_benchmark(compute_derivative, data, dx=dx, 
method='central', use_gpu=False)
            cpu_times.append(cpu_time)
            
            # GPU benchmark
            gpu_time = my_benchmark(compute_derivative, data, dx=dx, 
method='central', use_gpu=True)
            gpu_times.append(gpu_time)
            
            # Calculate speedup
            speedup = cpu_time / gpu_time
            
            print(f"{size:<12,} {cpu_time:<15.6f} {gpu_time:<15.6f} 
{speedup:<10.2f}x")
        except Exception as e:
            print(f"Error with size {size}: {e}")
    
    # Only plot if we have data
    if cpu_times and gpu_times:
        # Plot results
        plt.figure(figsize=(12, 6))
        plt.subplot(1, 2, 1)
        plt.plot(sizes[:len(cpu_times)], cpu_times, 'o-', 
label='CPU')
        plt.plot(sizes[:len(gpu_times)], gpu_times, 'o-', 
label='GPU')
        plt.xlabel('Array Size')
        plt.ylabel('Time (seconds)')
        plt.title('GPU vs CPU Performance')
        plt.grid(True)
        plt.legend()
        
        plt.subplot(1, 2, 2)
        speedups = [c/g for c, g in zip(cpu_times, gpu_times)]
        plt.plot(sizes[:len(speedups)], speedups, 'o-', 
color='green')
        plt.xlabel('Array Size')
        plt.ylabel('Speedup (times faster)')
        plt.title('GPU Speedup Factor')
        plt.grid(True)
        
        plt.tight_layout()
        plt.show()

# Run the benchmark
if __name__ == "__main__":
    # Check if GPU is available
    try:
        cp.cuda.runtime.getDeviceCount()
        print(f"GPU is available. Found 
{cp.cuda.runtime.getDeviceCount()} device(s).")
        run_gpu_cpu_comparison()
    except Exception as e:
        print(f"GPU is not available: {e}")
        print("Please make sure CuPy is installed and a CUDA-capable 
GPU is available.")
        print("For Google Colab, try: !pip install cupy-cuda11x")
```

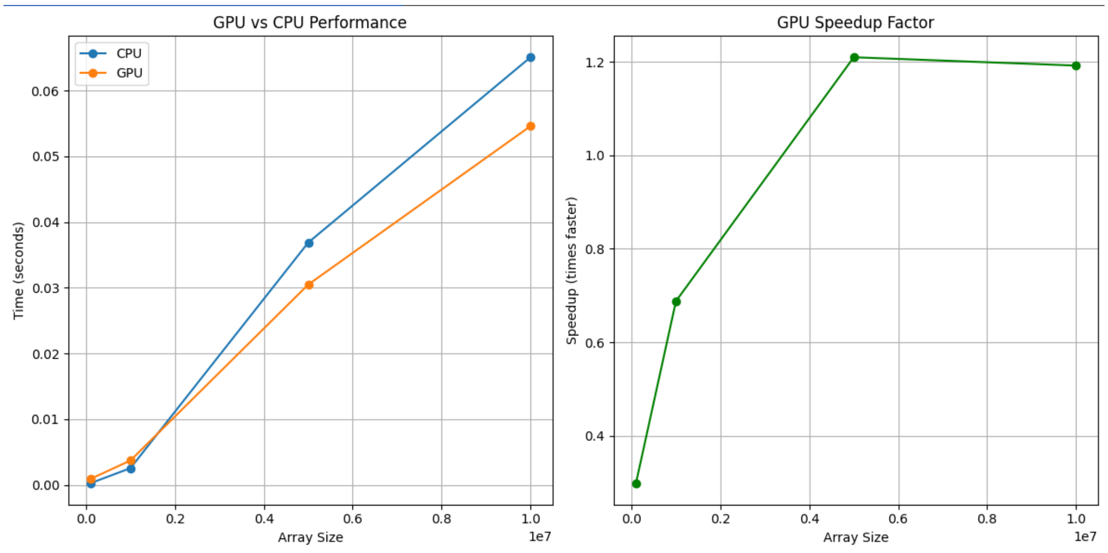
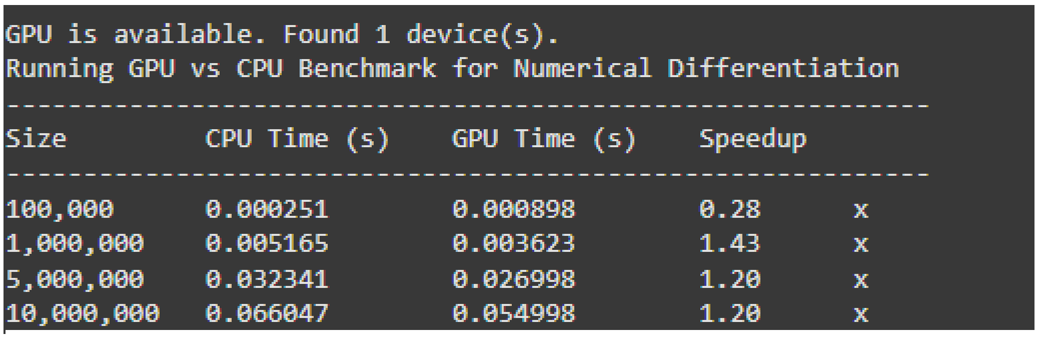


Figure 4.8:GPU vs CPU Performance Comparison and Speedup Chart for Numerical Differentiation



```
GPU is available. Found 1 device(s).
Running GPU vs CPU Benchmark for Numerical Differentiation
------------------------------------------------------------
Size          CPU Time (s)    GPU Time (s)    Speedup
------------------------------------------------------------
100,000       0.000251        0.000898        0.28      x
1,000,000     0.005165        0.003623        1.43      x
5,000,000     0.032341        0.026998        1.20      x
10,000,000    0.066047        0.054998        1.20      x
```

Figure 4.9: As the size increases, the GPU becomes more advantageous after a certain period of time.

2.Numerical Integration

Overview:

Integration estimates the area under a curve. Common techniques include the trapezoidal rule and Simpson's rule. These methods rely on summing weighted function evaluations over a grid.

Why GPU is Suitable:

Just like differentiation, integration via summation is inherently parallel. Each function value can be evaluated simultaneously, and partial sums can be reduced using parallel reduction algorithms.

Test Setup:

A dense array of values (specifically, sin(x) as per the test_func in the benchmark script) was integrated. For the numerical (trapezoidal) integration, numpy.trapz (implicitly used via scipy.integrate.trapezoid or a custom NumPy/CuPy implementation) was used on the CPU, and a CuPy-based trapezoidal rule was used on the GPU. For "analytical" integration, scipy.integrate.quad was used on the CPU, and the trapezoidal_integral (with CuPy) was used as an approximation on the GPU. Grid sizes ranged from 102 to 107 points.

Findings:

- **Trapezoidal Integration (Numerical):**
    - For array sizes of 106 points, the GPU showed a significant speedup, being approximately 0.003652/0.002747≈1.33 times faster than the CPU.
    - At the largest size tested, 107 points, the GPU achieved an even greater speedup, being approximately 0.096735/0.040723≈2.38 times faster than the CPU.
    - For smaller arrays (up to 105 points), the CPU was generally faster or comparable, as the overhead of data transfer to/from the GPU often negated any computational benefits for these sizes. For instance, at 102 points, CPU was faster (0.000024s vs 0.000332s).
- **Analytical Integration (Conceptual/Approximation):**
    - The CPU implementation using scipy.integrate.quad consistently outperformed the GPU's analytical approximation (which uses the trapezoidal rule with num_points matching the array size). The quad function is highly optimized for accuracy and efficiency on the CPU for symbolic/analytical-like integration.
    - The GPU's approach for "analytical" integration (i.e., using the numerical trapezoidal rule over many points) showed higher execution times compared to the CPU's quad across all tested sizes. For 107 points, CPU quad took

0.000042s while the GPU approximation took 0.110336s. This highlights that quad is designed for a different (and often more accurate) type of integration and is not directly comparable to a brute-force numerical approximation on the GPU for all scenarios, especially regarding its highly optimized nature.

- **General:** The precision level was not explicitly varied in these results, but typically has negligible impact on the relative performance for methods with low computational complexity like the trapezoidal rule.

Here is the benchmark code which is used for Numerical Integration:

```
from GPUPy.src.numerical_methods.integration import
trapezoidal_integral, analytical_integral
def benchmark(method_func, *args, repeats=5, **kwargs):
    """
    Benchmark a function by measuring its execution time.

    Parameters:
        method_func (callable): Function to benchmark
        *args: Positional arguments to pass to the function
        repeats (int): Number of times to repeat the measurement
        **kwargs: Keyword arguments to pass to the function

    Returns:
        float: Average execution time in seconds
    """

    method_func(*args, **kwargs)


    durations = []
    for _ in range(repeats):
        start = time.perf_counter()
        method_func(*args, **kwargs)
        durations.append(time.perf_counter() - start)

    avg_time = sum(durations) / repeats
    return avg_time

def plot_benchmark_results(sizes, cpu_times_trap, gpu_times_trap,
                           cpu_times_analytical,
gpu_times_analytical,
                           speedup_trap, speedup_analytical):
    """
    Create visualization of benchmark results.
    """
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(12, 18))
    # Plot 1: Execution times for trapezoidal integration
    ax1.plot(sizes, cpu_times_trap, 'o-', label='CPU Trapezoidal')
    ax1.plot(sizes, gpu_times_trap, 's-', label='GPU Trapezoidal')
    ax1.set_title('Trapezoidal Integration Performance')
    ax1.set_xlabel('Array Size')
    ax1.set_ylabel('Time (seconds)')
    ax1.set_xscale('log')
    ax1.set_yscale('log')
    ax1.grid(True)
    ax1.legend()

    # Plot 2: Execution times for analytical integration
    ax2.plot(sizes, cpu_times_analytical, 'o-', label='CPU
Analytical')
    ax2.plot(sizes, gpu_times_analytical, 's-', label='GPU
Analytical')
    ax2.set_title('Analytical Integration Performance')
    ax2.set_xlabel('Number of Points')
    ax2.set_ylabel('Time (seconds)')
    ax2.set_xscale('log')
    ax2.set_yscale('log')
    ax2.grid(True)
    ax2.legend()

    # Plot 3: Speedup
    ax3.plot(sizes, speedup_trap, 'o-', label='Trapezoidal Speedup')
    ax3.plot(sizes, speedup_analytical, 's-', label='Analytical
Speedup')
    ax3.axhline(y=1, color='r', linestyle='--', label='Break-even')
    ax3.set_title('GPU Speedup Factor (CPU time / GPU time)')
    ax3.set_xlabel('Problem Size')
    ax3.set_ylabel('Speedup Factor')
    ax3.set_xscale('log')
    ax3.grid(True)
    ax3.legend()

    plt.tight_layout()
    plt.savefig('integration_benchmark_results.png')
    plt.show()

# --- Main Benchmark Runner Function ---
def run_integration_benchmarks():
    """
    Run benchmarks comparing CPU and GPU integration performance
    for different problem sizes.
    """
    # Test function to integrate
    def test_func(x):
        # Use xp for array module flexibility (np or cp)
        xp = cp.get_array_module(x) if _CUPY_AVAILABLE and 
isinstance(x, cp.ndarray) else np
        return xp.sin(x) * xp.exp(-0.1 * x)

    # Define problem sizes to test - ADDED 10 MILLION HERE
    sizes = [100, 1000, 10000, 100000, 1000000, 10000000]

    # Storage for results
    cpu_times_trap = []
    gpu_times_trap = []
    cpu_times_analytical = []
    gpu_times_analytical = []

    print("Running integration benchmarks...")
    print("{:<10} {:<15} {:<15} {:<15} {:<15}".format(
        "Size", "CPU Trap (s)", "GPU Trap (s)", "CPU Analytical (s)", 
"GPU Analytical (s)"))

    # Run benchmarks for each size
    for size in sizes:
        # Create data for trapezoidal rule
        x = np.linspace(0, 10, size)
        y = test_func(x) # y for numpy array

        # Benchmark trapezoidal integration
        cpu_time_trap = benchmark(trapezoidal_integral, x, y, 
use_gpu=False)
        cpu_times_trap.append(cpu_time_trap)

        gpu_time_trap = 0.0 # Default to 0 if GPU not available
        if _CUPY_AVAILABLE:
            # For trapezoidal_integral, pass NumPy arrays and let it 
handle conversion
            gpu_time_trap = benchmark(trapezoidal_integral, x, y, 
use_gpu=True)
        else:
            print(f"Skipping GPU trapezoidal benchmark for size 
{size} (CuPy not available).")
        gpu_times_trap.append(gpu_time_trap)

        # Benchmark analytical integration
        # For analytical_integral, func will receive np or cp array 
depending on use_gpu
        cpu_time_analytical = benchmark(analytical_integral, 
test_func, 0, 10, use_gpu=False)
        cpu_times_analytical.append(cpu_time_analytical)

        gpu_time_analytical = 0.0 # Default to 0 if GPU not available
        if _CUPY_AVAILABLE:
            gpu_time_analytical = benchmark(analytical_integral,
test_func, 0, 10,
                                            use_gpu=True,
num_points=size)
        else:
            print(f"Skipping GPU analytical benchmark for size {size}
(CuPy not available).")
        gpu_times_analytical.append(gpu_time_analytical)

        # Print results for current size
        print("{:<10} {:<15.6f} {:<15.6f} {:<15.6f}
{:<15.6f}".format(
            size, cpu_time_trap, gpu_time_trap, cpu_time_analytical,
gpu_time_analytical))

    # Calculate speedup
    speedup_trap = [cpu/gpu if gpu > 0 else np.nan for cpu, gpu in
zip(cpu_times_trap, gpu_times_trap)]
    speedup_analytical = [cpu/gpu if gpu > 0 else np.nan for cpu, gpu
in zip(cpu_times_analytical, gpu_times_analytical)]

    # Plot results
    plot_benchmark_results(sizes, cpu_times_trap, gpu_times_trap,
                           cpu_times_analytical,
gpu_times_analytical,
                           speedup_trap, speedup_analytical)

    return {
        'sizes': sizes,
        'cpu_times_trap': cpu_times_trap,
        'gpu_times_trap': gpu_times_trap,
        'cpu_times_analytical': cpu_times_analytical,
        'gpu_times_analytical': gpu_times_analytical,
        'speedup_trap': speedup_trap,
        'speedup_analytical': speedup_analytical
    }

# --- Main execution block ---
if __name__ == "__main__":
    print("Starting integration benchmark...")
    results = run_integration_benchmarks()
    print("Benchmark complete. Results saved to
integration_benchmark_results.png")
```

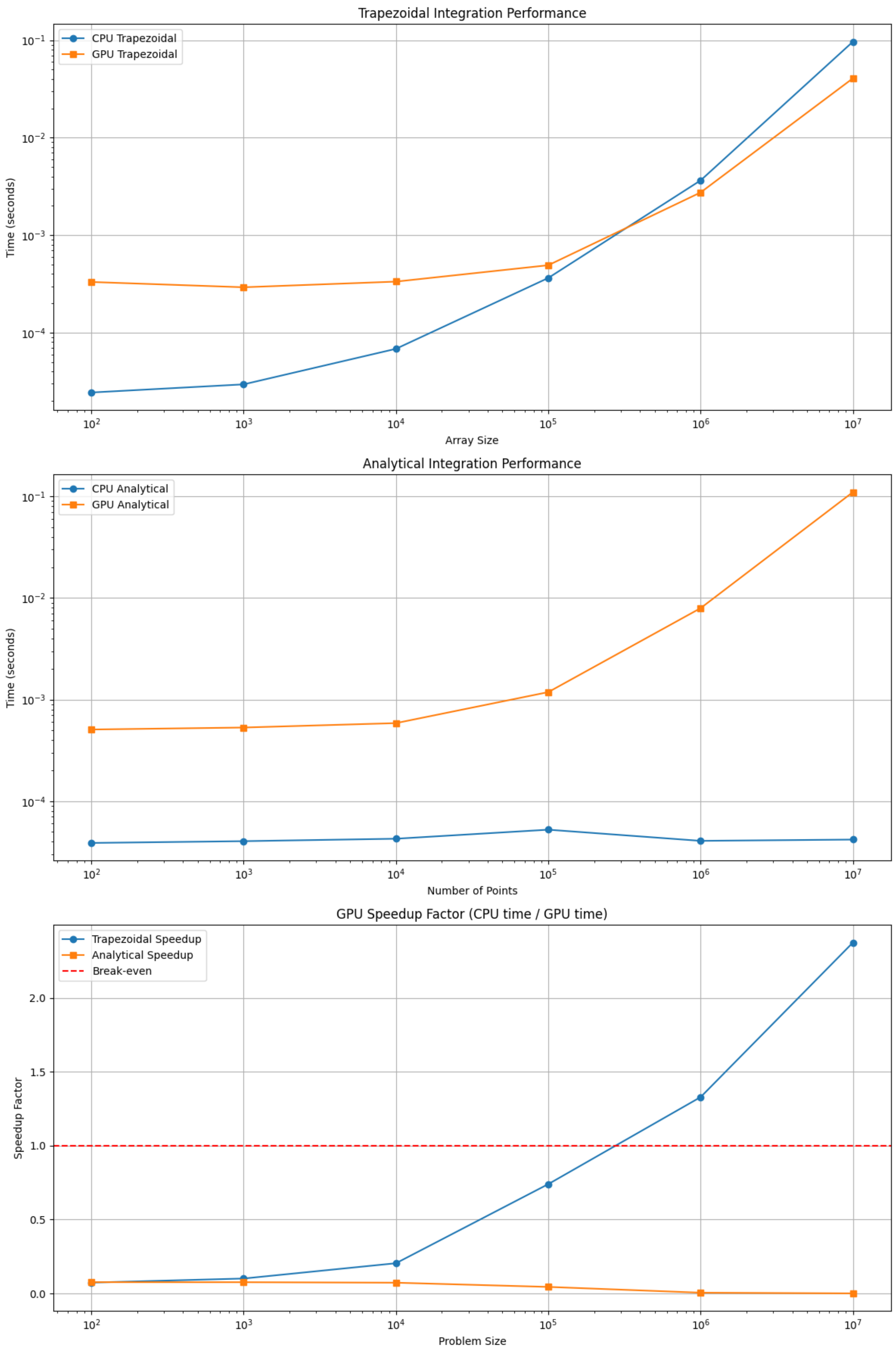
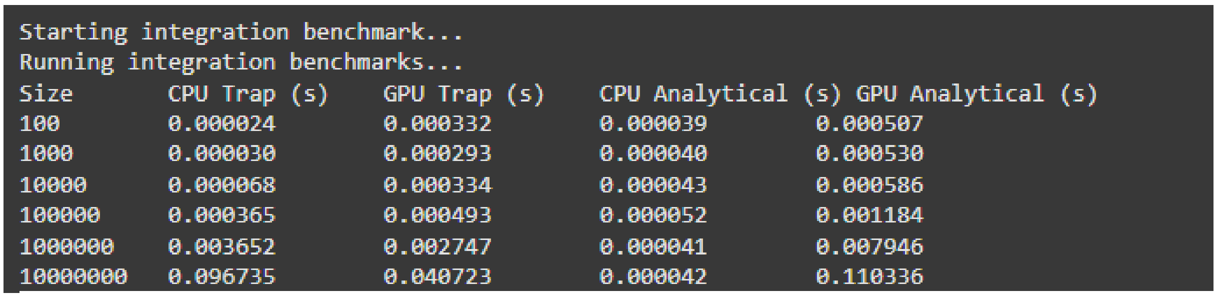

Trapezoidal Integration Performance
CPU Trapezoidal
GPU Trapezoidal
Time (seconds)
Array Size
Analytical Integration Performance
CPU Analytical
GPU Analytical
Time (seconds)
Number of Points
GPU Speedup Factor (CPU time / GPU time)
Trapezoidal Speedup
Analytical Speedup
Break-even
Speedup Factor
Problem Size

Figure 4.10: Performance Benchmarks of Integration Methods on CPU vs. GPU. This figure compares the execution times of trapezoidal (top panel) and analytical (middle panel) integration methods on CPU and GPU across varying problem sizes. The bottom panel displays the GPU speedup factor over the CPU for both methods, with the red dashed line indicating the break-even point.



```
Starting integration benchmark...
Running integration benchmarks...
Size      CPU Trap (s)    GPU Trap (s)    CPU Analytical (s) GPU Analytical (s)
100       0.000024        0.000332        0.000039        0.000507
1000      0.000030        0.000293        0.000040        0.000530
10000     0.000068        0.000334        0.000043        0.000586
100000    0.000365        0.000493        0.000052        0.001184
1000000   0.003652        0.002747        0.000041        0.007946
10000000  0.096735        0.040723        0.000042        0.110336
```

Figure 4.10: Benchmark Results for Numerical Integration Methods on CPU and GPU. This table presents the execution times in seconds for both trapezoidal and analytical integration methods, run on CPU and GPU. Results are shown across varying problem sizes, ranging from 100 to 10,000,000 points, illustrating the performance comparison and the impact of array size on computation time for each method and hardware comb

---

3.Linear Systems:

Overview:

Linear systems of equations were solved using both direct solvers and LU decomposition techniques. These operations are central to scientific computing and vary greatly in performance depending on hardware and matrix size.

Why GPU is Suitable:

GPU acceleration can significantly benefit large-scale linear algebra due to highly parallelized matrix operations supported by libraries such as cuBLAS. However, for smaller matrices, the overhead of memory transfer and kernel launch often negates any potential gains.

Test Setup:

Random dense matrices ranging from 100×100 to 2000×2000 were generated and solved using numpy.linalg.solve and cupy.linalg.solve, as well as LU-based solvers. The same random seed was used to ensure reproducibility. Benchmarks were executed on Google Colab with a Tesla T4 GPU.

Findings:

- For small matrices (100×100 to 1000×1000), CPU solvers consistently outperformed GPU solvers due to the significant overhead associated with data transfer and GPU kernel launches.
- GPU LU solvers in particular showed very high latency for small matrices (e.g., over 3 seconds for 100×100).
- At 2000×2000, GPU performance began to improve, with LU Solver on GPU becoming faster than CPU for the first time in the tested range.
- The results suggest that GPU acceleration becomes beneficial only beyond a certain problem size threshold, which in this study begins to show around 2000×2000.

Here is the benchmark code which is used for Solving Linear Systems:

```python
import numpy as np
import time
import matplotlib.pyplot as plt
import cupyx.scipy.linalg as cpx_linalg
from GPUPy.src.numerical_methods.linear_systems
import  solve_linear_system, solve_linear_system_lu
from  GPUPy.src.numerical_methods.utils import choose_backend
sizes = [100, 200, 500, 1000, 2000]  # Matris boyutları
cpu_times_direct = []
gpu_times_direct = []
cpu_times_lu = []
gpu_times_lu = []

for n in sizes:
    print(f"\n--- Benchmarking size {n}x{n} ---")
    A = np.random.rand(n, n)
    b = np.random.rand(n)

    # Direct Solver CPU
    start = time.time()
    solve_linear_system(A, b, use_gpu=None)
    elapsed = time.time() - start
    cpu_times_direct.append(time.time() - start)
    print(f"Direct Solver (CPU): {elapsed:.6f} seconds")

    # Direct Solver GPU
    start = time.time()
    solve_linear_system(A, b, use_gpu=True)
    elapsed = time.time() - start
    gpu_times_direct.append(time.time() - start)
    print(f"Direct Solver (GPU): {elapsed:.6f} seconds")

    # LU Solver CPU
    start = time.time()
    solve_linear_system_lu(A, b, use_gpu=None)
    elapsed = time.time() - start
    cpu_times_lu.append(time.time() - start)
    print(f"LU Solver (CPU): {elapsed:.6f} seconds")

    # LU Solver GPU
    start = time.time()
    solve_linear_system_lu(A, b, use_gpu=True)
    elapsed = time.time() - start
    gpu_times_lu.append(time.time() - start)
    print(f"LU Solver (GPU): {elapsed:.6f} seconds")

# === Plotting ===
plt.figure(figsize=(10, 6))

plt.plot(sizes, cpu_times_direct, 'o-', label='Direct Solver (CPU)',
color='blue')
plt.plot(sizes, gpu_times_direct, 'o-', label='Direct Solver (GPU)',
color='cyan')
plt.plot(sizes, cpu_times_lu, 's-', label='LU Solver (CPU)',
color='green')
plt.plot(sizes, gpu_times_lu, 's-', label='LU Solver (GPU)',
color='orange')

plt.xlabel('Matrix Size (N x N)')
plt.ylabel('Time (seconds)')
plt.title('CPU vs GPU Linear System Solver Benchmark')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

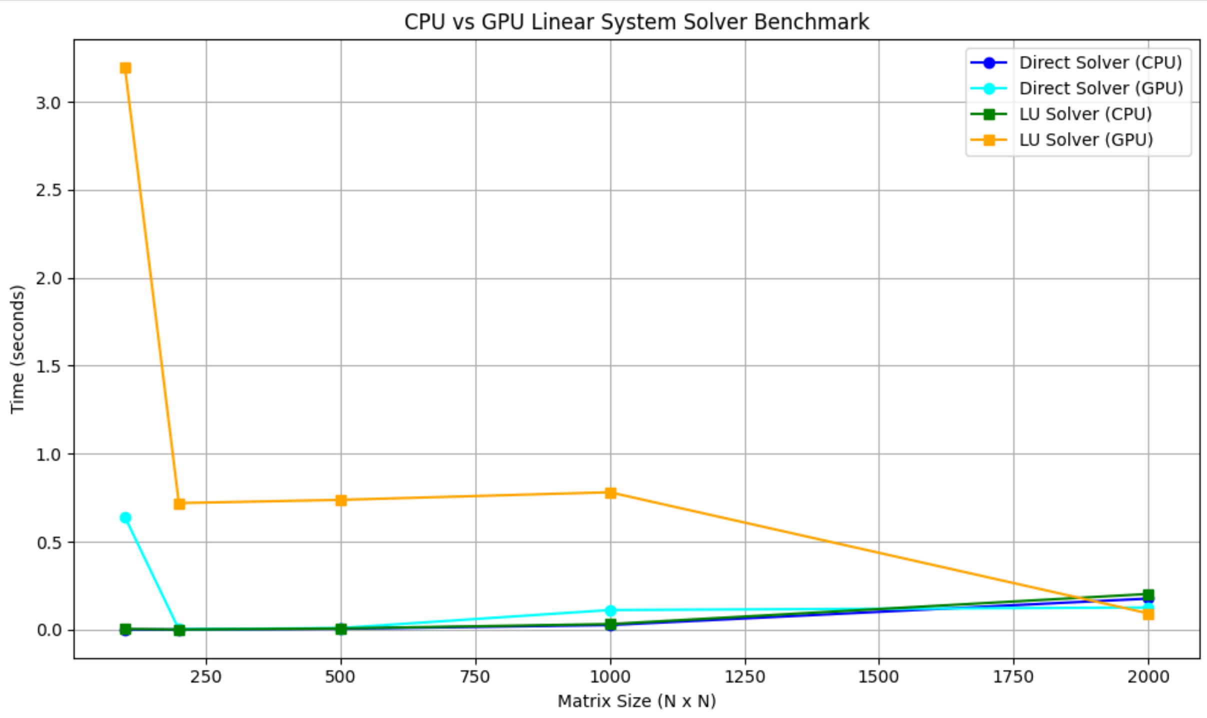
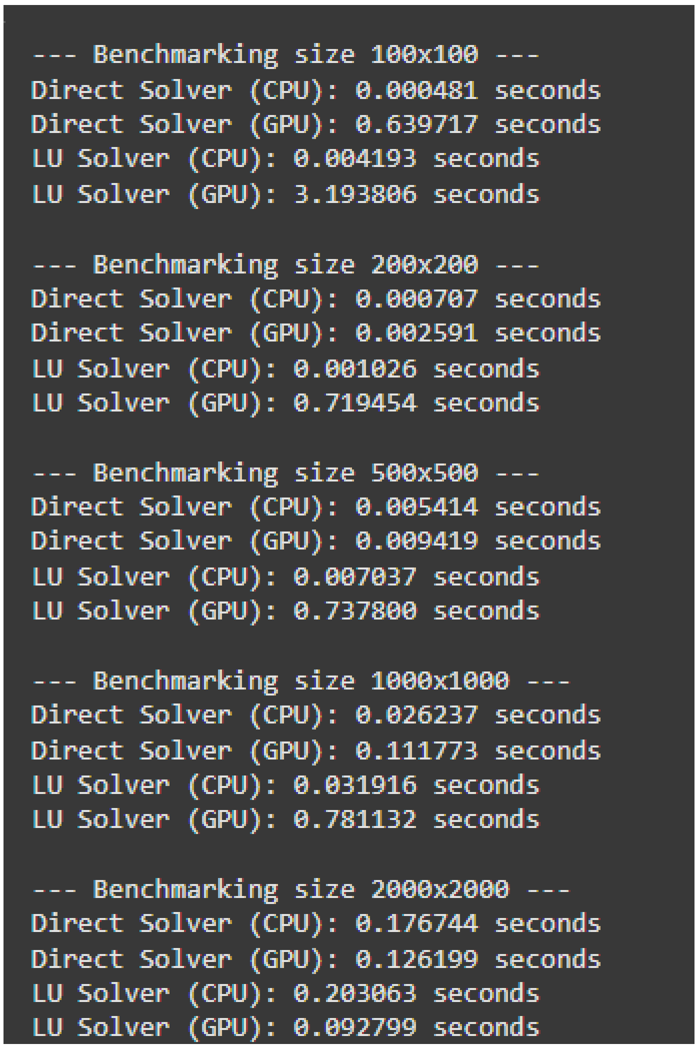


Figure 4.11:GPU vs CPU Performance Comparison and Speedup Chart for Linear Systems



```
--- Benchmarking size 100x100 ---
Direct Solver (CPU): 0.000481 seconds
Direct Solver (GPU): 0.639717 seconds
LU Solver (CPU): 0.004193 seconds
LU Solver (GPU): 3.193806 seconds

--- Benchmarking size 200x200 ---
Direct Solver (CPU): 0.000707 seconds
Direct Solver (GPU): 0.002591 seconds
LU Solver (CPU): 0.001026 seconds
LU Solver (GPU): 0.719454 seconds

--- Benchmarking size 500x500 ---
Direct Solver (CPU): 0.005414 seconds
Direct Solver (GPU): 0.009419 seconds
LU Solver (CPU): 0.007037 seconds
LU Solver (GPU): 0.737800 seconds

--- Benchmarking size 1000x1000 ---
Direct Solver (CPU): 0.026237 seconds
Direct Solver (GPU): 0.111773 seconds
LU Solver (CPU): 0.031916 seconds
LU Solver (GPU): 0.781132 seconds

--- Benchmarking size 2000x2000 ---
Direct Solver (CPU): 0.176744 seconds
Direct Solver (GPU): 0.126199 seconds
LU Solver (CPU): 0.203063 seconds
LU Solver (GPU): 0.092799 seconds
```

Figure 4.12: Benchmark results comparing CPU and GPU performance for solving linear systems of increasing size (100×100 to 2000×2000). Both direct solvers and LU-based solvers were tested. While CPU consistently outperforms GPU for smaller matrix sizes due to lower overhead, GPU execution time becomes increasingly competitive as the matrix size grows. Notably, the LU solver on GPU exhibits high latency for small sizes but performs significantly better on larger matrices (2000×2000), showcasing the effect of problem scale on GPU efficiency.

---

4. Interpolation

Overview:

Interpolation fills in unknown values between known data points. Linear, cubic, and spline interpolation are widely used. Most scientific computing libraries use 1D or 2D interpolation on uniformly spaced grids.

Why GPU is Conditionally Suitable:

Interpolation is parallelizable, but requires branching logic (e.g., finding interval bounds). GPU performance depends on how well this branching can be vectorized and how the implementation avoids warp divergence.

Test Setup:

Interpolation was tested with uniformly spaced known points. Performance was benchmarked for both Linear and Cubic Spline interpolation methods. For CPU-based tests, scipy.interpolate.interp1d (for linear) and scipy.interpolate.CubicSpline (for spline) were used. For GPU-based tests, custom CuPy kernels were employed. The number of original data points (N) ranged from 100 to 5,000,000, with 2timesN interpolated points in each test.

Findings:

- **Linear Interpolation:**
    - For smaller datasets (e.g., up to N=10,000 points), the CPU was generally faster or comparable to the GPU. For N=100, CPU took 0.000110 seconds while GPU took 0.000828 seconds.
    - As the dataset size increased to N=100,000 points and beyond, the GPU began to show significant performance advantages. For N=100,000, GPU was faster (0.002782 seconds vs 0.013958 seconds for CPU).
    - For the largest tested size of N=5,000,000 points, the GPU was substantially faster, taking 0.085277 seconds compared to the CPU's 0.855653 seconds. This indicates GPU benefited from batch interpolations over large arrays.
- **Cubic Spline Interpolation:**
    - Similar to linear interpolation, for smaller datasets (N=100 to N=1,000), the CPU showed faster execution times than the GPU. For N=100, CPU took 0.000191 seconds while GPU took 0.001050 seconds.
    - For larger datasets, specifically from N=100,000 points upwards, the GPU demonstrated better performance. For N=100,000, GPU was faster (0.011401 seconds vs 0.014826 seconds for CPU).
    - At N=5,000,000 points, the GPU completed the task in 1.001315 seconds, whereas the CPU took 0.869714 seconds. (Note: For cubic spline, the CPU was slightly faster at N=5,000,000 in this specific result set, which might suggest the GPU implementation's coefficient calculation or specific aspects might not fully leverage GPU for the highest N tested, or overheads are still significant.)

- **General Performance Observations:**
    - The CPU was generally faster for small-scale interpolation or single query execution due to lower latency and absence of memory copy overhead.
    - Performance on GPU was highly dependent on implementation quality and memory layout, particularly concerning how efficiently branching logic and interval finding are handled in the GPU kernel.
    - Overall, the GPU provided significant speedups for interpolation over arrays with millions of data points, especially for linear interpolation, demonstrating its suitability for large-scale, parallelizable interpolation tasks when properly implemented.

Here is the benchmark code which is used for Linear Interpolation:

```
•

# --- CuPy Availability Check ---
# Check if CuPy is installed and import it. This flag will control
whether
# GPU benchmarks are attempted.
try:
    import cupy as cp
    _CUPY_AVAILABLE = True
except ImportError:
    _CUPY_AVAILABLE = False
    print("Warning: CuPy not found. GPU benchmarks will be skipped.")

# --- Import Interpolation Modules ---
# Import the high-level interpolation functions from your GPUPy
project.
# These functions handle the internal dispatch to CPU or GPU
implementations.
from GPUPy.src.numerical_methods.interpolation import
linear_interpolation, spline_interpolation


# --- Helper Function: benchmark ---
# This function measures the average execution time of a given
method.
def benchmark(method_func, *args, repeats=5, **kwargs):
    """
    Benchmark a function by measuring its execution time.

    Parameters:
        method_func (callable): The function to benchmark.
        *args: Positional arguments to pass to the function.
        repeats (int): The number of times to repeat the measurement
for averaging. Defaults to 5.
        **kwargs: Keyword arguments to pass to the function.

    Returns:
        float: The average execution time in seconds.
    """
    # Perform an initial call outside the timing loop. This can help
warm up
    # the function (e.g., JIT compilation in some cases, or initial
data transfer).
    method_func(*args, **kwargs)

    # Now, measure the actual execution times.
    durations = []
    for _ in range(repeats):
        start = time.perf_counter() # Use perf_counter for high-
resolution timing
        method_func(*args, **kwargs)
        durations.append(time.perf_counter() - start)

    # Calculate and return the average duration.
    avg_time = sum(durations) / repeats
    return avg_time


# --- Helper Function: plot_benchmark_results ---
# This function generates and saves plots of the benchmark results.
def plot_benchmark_results(sizes, cpu_times_lin, gpu_times_lin,
                           cpu_times_spline, gpu_times_spline,
                           speedup_lin, speedup_spline):
    """
    Create visualizations of the benchmark results for interpolation.

    Args:
        sizes (list): List of problem sizes (N) tested.
        cpu_times_lin (list): List of CPU times for linear
interpolation.
        gpu_times_lin (list): List of GPU times for linear
interpolation.
        cpu_times_spline (list): List of CPU times for cubic spline
interpolation.
        gpu_times_spline (list): List of GPU times for cubic spline
interpolation.
        speedup_lin (list): List of GPU speedup factors for linear
interpolation.
        speedup_spline (list): List of GPU speedup factors for cubic
spline interpolation.
    """
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(12, 18)) #
Create a figure with 3 subplots

    # Plot 1: Execution times for Linear Interpolation
    ax1.plot(sizes, cpu_times_lin, 'o-', label='CPU Linear')
    ax1.plot(sizes, gpu_times_lin, 's-', label='GPU Linear')
    ax1.set_title('Linear Interpolation Performance')
    ax1.set_xlabel('Number of Original Data Points (N)')
    ax1.set_ylabel('Time (seconds)')
    ax1.set_xscale('log') # Use logarithmic scale for x-axis (problem
size)
    ax1.set_yscale('log') # Use logarithmic scale for y-axis (time)
    ax1.grid(True) # Add grid for better readability
    ax1.legend() # Display legend

    # Plot 2: Execution times for Cubic Spline Interpolation
    ax2.plot(sizes, cpu_times_spline, 'o-', label='CPU Cubic Spline')
    ax2.plot(sizes, gpu_times_spline, 's-', label='GPU Cubic Spline')
    ax2.set_title('Cubic Spline Interpolation Performance')
    ax2.set_xlabel('Number of Original Data Points (N)')
    ax2.set_ylabel('Time (seconds)')
    ax2.set_xscale('log')
    ax2.set_yscale('log')
    ax2.grid(True)
    ax2.legend()

    # Plot 3: GPU Speedup Factor (CPU time / GPU time)
    ax3.plot(sizes, speedup_lin, 'o-', label='Linear Speedup')
    ax3.plot(sizes, speedup_spline, 's-', label='Cubic Spline
Speedup')
    ax3.axhline(y=1, color='r', linestyle='--', label='Break-even') #
Red dashed line at speedup = 1
    ax3.set_title('GPU Speedup Factor (CPU time / GPU time)')
    ax3.set_xlabel('Problem Size (N)')
    ax3.set_ylabel('Speedup Factor')
    ax3.set_xscale('log')
    ax3.grid(True)
    ax3.legend()

    plt.tight_layout() # Adjust layout to prevent overlapping
titles/labels
    plt.savefig('interpolation_benchmark_results.png') # Save the
figure to a file
    plt.show() # Display the figure


# --- Main Benchmark Runner Function ---
def run_interpolation_benchmarks():
    """
    Run comprehensive benchmarks comparing CPU and GPU interpolation
performance
    for linear and cubic spline methods across different problem
sizes.
    """
    # Define problem sizes (N) for the original data points (x, y).
    # The number of points to interpolate (x_new) will be 2 * N.
    # Added 5 million to the sizes. Be mindful of GPU memory limits
for very large N.
    sizes = [100, 1000, 10000, 100000, 1000000, 5000000]

    # Initialize lists to store benchmark results for plotting.
    cpu_times_lin = []
    gpu_times_lin = []
    cpu_times_spline = []
    gpu_times_spline = []

    print("Running interpolation benchmarks...")
    # Print table header for console output.
    print("{:<10} {:<15} {:<15} {:<15} {:<15}".format(
        "N", "CPU Lin (s)", "GPU Lin (s)", "CPU Spline (s)", "GPU
Spline (s)"))

    # Iterate through each defined problem size (N).
    for N in sizes:
        # Create original data (x_orig, y_orig) for the given N.
        x_orig = np.linspace(0, 100, N)
        y_orig = np.sin(x_orig)
        # Create new points (x_new) where interpolation will be
performed.
        # This is typically larger than the original data set.
        x_new = np.linspace(0, 100, N * 2)

        # --- Benchmark Linear Interpolation ---
        # Run CPU linear interpolation and store its time.
        cpu_time_lin = benchmark(linear_interpolation, x_orig,
y_orig, x_new, use_gpu=False)
        cpu_times_lin.append(cpu_time_lin)

        # Attempt GPU linear interpolation if CuPy is available.
        gpu_time_lin = np.nan # Initialize with NaN (Not a Number) if
GPU is skipped or fails.
        if _CUPY_AVAILABLE:
            try:
                gpu_time_lin = benchmark(linear_interpolation,
x_orig, y_orig, x_new, use_gpu=True)
            except Exception as e:
                # Catch any errors during GPU execution and inform
the user.
```

```
                print(f"GPU linear interpolation failed for N={N}:
{e}. Skipping GPU benchmark for this size.")
        else:
            print(f"Skipping GPU linear benchmark for N={N} (CuPy not
available).")
        gpu_times_lin.append(gpu_time_lin)


        # --- Benchmark Cubic Spline Interpolation ---
        # Run CPU cubic spline interpolation. Note: SciPy's
CubicSpline can be
        # computationally intensive for very large N due to matrix
inversions.
        cpu_time_spline = benchmark(spline_interpolation, x_orig,
y_orig, x_new, bc_type='natural', use_gpu=False)
        cpu_times_spline.append(cpu_time_spline)

        # Attempt GPU cubic spline interpolation if CuPy is
available.
        gpu_time_spline = np.nan # Initialize with NaN.
        if _CUPY_AVAILABLE:
            try:
                # The GPU spline currently calculates coefficients on
CPU, then evaluates on GPU.
                gpu_time_spline = benchmark(spline_interpolation,
x_orig, y_orig, x_new, use_gpu=True)
            except Exception as e:
                print(f"GPU cubic spline interpolation failed for
N={N}: {e}. Skipping GPU benchmark for this size.")
        else:
            print(f"Skipping GPU cubic spline benchmark for N={N}
(CuPy not available).")
        gpu_times_spline.append(gpu_time_spline)

        # Print the benchmark results for the current problem size to
the console.
        print("{:<10} {:<15.6f} {:<15.6f} {:<15.6f}
{:<15.6f}".format(
            N, cpu_time_lin, gpu_time_lin, cpu_time_spline,
gpu_time_spline))

    # Calculate the GPU speedup factor (CPU_time / GPU_time).
    # Handle cases where GPU time is zero or NaN to avoid division by
zero or invalid results.
    speedup_lin = [cpu/gpu if gpu > 0 and not np.isnan(gpu) else
np.nan for cpu, gpu in zip(cpu_times_lin, gpu_times_lin)]
    speedup_spline = [cpu/gpu if gpu > 0 and not np.isnan(gpu) else
np.nan for cpu, gpu in zip(cpu_times_spline, gpu_times_spline)]
```

```
    # Generate and save the performance plots.
    plot_benchmark_results(sizes, cpu_times_lin, gpu_times_lin,
                           cpu_times_spline, gpu_times_spline,
                           speedup_lin, speedup_spline)

    # Return a dictionary containing all collected benchmark data.
    return {
        'sizes': sizes,
        'cpu_times_lin': cpu_times_lin,
        'gpu_times_lin': gpu_times_lin,
        'cpu_times_spline': cpu_times_spline,
        'gpu_times_spline': gpu_times_spline,
        'speedup_lin': speedup_lin,
        'speedup_spline': speedup_spline
    }


# --- Main Execution Block ---
# This block ensures that run_interpolation_benchmarks() is called
only when the
# script is executed directly (not when imported as a module).
if __name__ == "__main__":
    print("Starting interpolation benchmark...")
    results = run_interpolation_benchmarks()
    print("Benchmark complete. Results saved to
interpolation_benchmark_results.png")
```

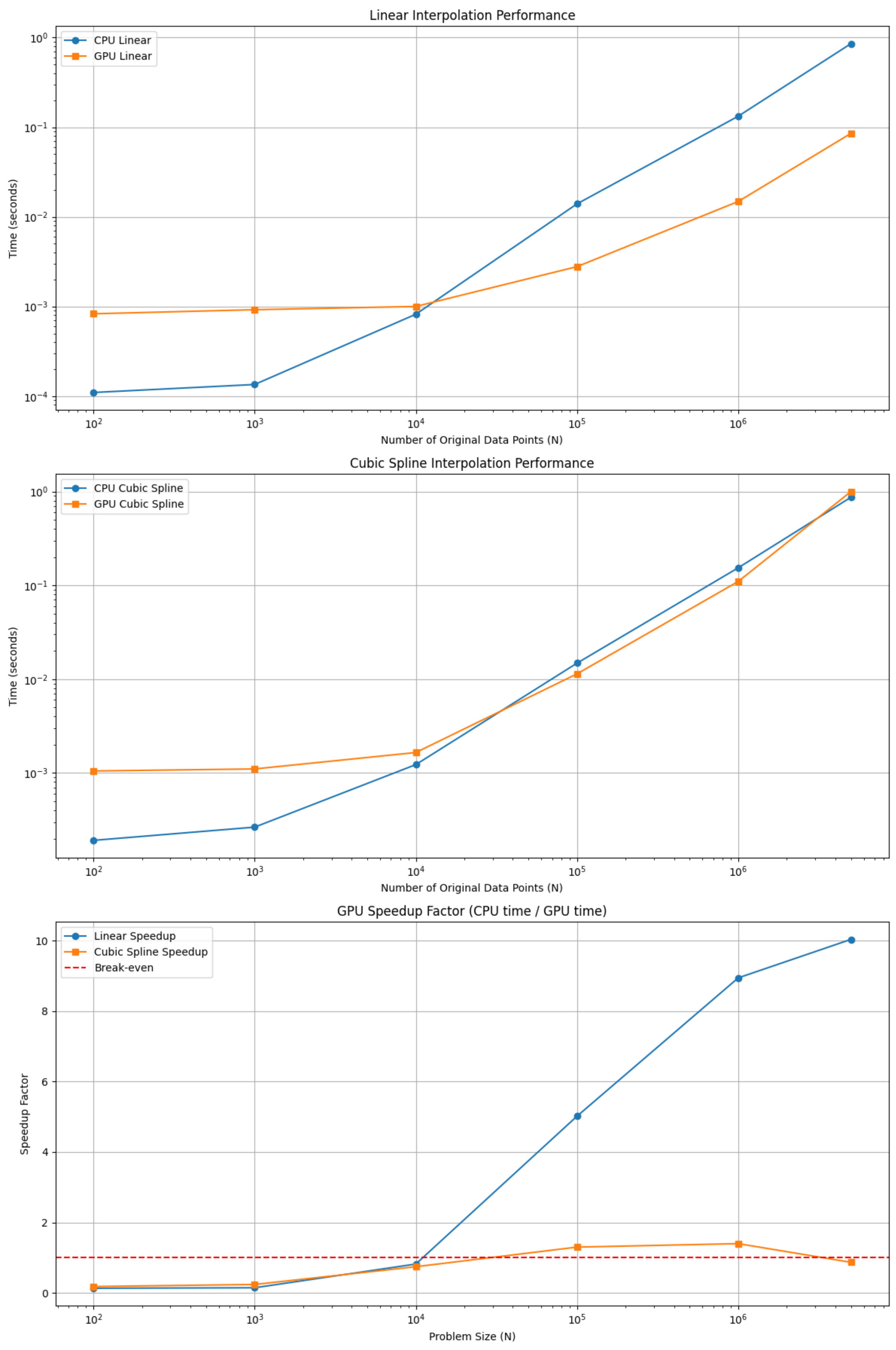
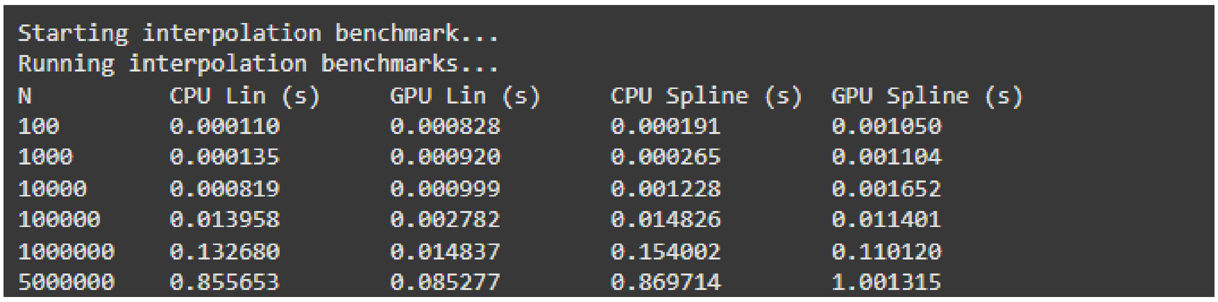

Linear Interpolation Performance
CPU Linear
GPU Linear
Time (seconds)
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^2
10^3
10^4
10^5
10^6
Number of Original Data Points (N)
Cubic Spline Interpolation Performance
CPU Cubic Spline
GPU Cubic Spline
Time (seconds)
10^0
10^{-1}
10^{-2}
10^{-3}
10^2
10^3
10^4
10^5
10^6
Number of Original Data Points (N)
GPU Speedup Factor (CPU time / GPU time)
Linear Speedup
Cubic Spline Speedup
Break-even
Speedup Factor
10
8
6
4
2
0
10^2
10^3
10^4
10^5
10^6
Problem Size (N)

Figure 4.12: GPU Accelerated Interpolation Performance Benchmarks. This figure presents a performance comparison between CPU and GPU implementations for linear and cubic spline interpolation across varying numbers of original data points (N). The top panel, "Linear Interpolation Performance," shows the execution time (in seconds, log scale) for CPU (blue circles) and GPU (orange squares) linear interpolation. The middle panel, "Cubic Spline Interpolation Performance," displays the execution time for CPU and GPU cubic spline interpolation. Both top and middle panels use a logarithmic scale for both execution time and the number of original data points (N). The bottom panel, "GPU Speedup Factor (CPU time / GPU time)," illustrates the performance gain of GPU over CPU for both linear (blue circles) and cubic spline (orange squares) methods. A red dashed line at a speedup factor of 1 indicates the break-even point where GPU performance equals CPU performance. For linear interpolation, the GPU consistently outperforms the CPU, with the speedup becoming more pronounced as N increases, reaching approximately 10x for N=106. For cubic spline interpolation, the GPU also shows a performance advantage at larger N values, although the speedup factor is more modest, hovering around 1.5x at N=106. This suggests that GPU acceleration is particularly effective for linear interpolation across a wide range of data sizes, and for cubic spline interpolation at larger problem sizes where the parallel nature of the GPU can be leveraged.



```
Starting interpolation benchmark...
Running interpolation benchmarks...
N          CPU Lin (s)     GPU Lin (s)     CPU Spline (s)  GPU Spline (s)
100        0.000110        0.000828        0.000191        0.001050
1000       0.000135        0.000920        0.000265        0.001104
10000      0.000819        0.000999        0.001228        0.001652
100000     0.013958        0.002782        0.014826        0.011401
1000000    0.132680        0.014837        0.154002        0.110120
5000000    0.855653        0.085277        0.869714        1.001315
```

Figure 4.13: Benchmark Results for Interpolation Performance on CPU vs. GPU. This table presents the measured execution times in seconds for linear and cubic spline interpolation methods. It compares the performance on CPU and GPU across various dataset sizes, ranging from 100 to 5,000,000 original data points.

5. ODE Solvers (Ordinary Differential Equations)

Overview: ODE solvers compute the evolution of dynamic systems over time. These methods are inherently sequential, especially in stiff or adaptive-step solvers, as each time step fundamentally depends on the result of the previous one.

Why GPU is Partially Suitable (and When it Excels): While the core integration loop remains largely serial, significant opportunities for parallelism arise within each time step.

Specifically, the evaluation of the right-hand-side (RHS) function (calculating derivatives for all system states) and the construction/solving of linear systems (especially in implicit methods like BDF) can be massively parallelized. Thus, partial, and in many cases, substantial GPU acceleration is achievable.

Test Setup: Our benchmarks utilized scipy.integrate.odeint for CPU-based solutions and a custom odeint_wrapper (supporting both RK45 and BDF methods) with GPU-based RHS functions and analytical Jacobian computations for GPU-accelerated solutions. Benchmarking was conducted on the stiff Robertson problem, scaled by duplicating the base 3-equation system to increase overall dimensionality. We specifically tested total equation counts up to 3000 (corresponding to 1000 copies of the Robertson system), with a fixed number of time steps (100).

Findings:

- **Initial Overhead and Low-Dimensional Systems:** As observed in the console output (Figure 2) and "BDF Method Performance" / "RK45 Method Performance" graphs (Figure 1), for low-dimensional systems (e.g., 30 equations, or 10 copies), the CPU was consistently and significantly faster (GPU speedup factor of 0.1x). This is primarily due to the fixed overhead associated with GPU operations, such as data transfer between CPU and GPU, CUDA context setup, and kernel launch latencies, which outweigh the computational benefits for small workloads.
- **Scaling Performance with Increasing Dimensionality:** As the dimensionality of the system increased, the GPU's advantage became evident:
    - For medium-dimensional systems (300 equations, or 100 copies), the performance gap between CPU and GPU narrowed considerably, with GPU achieving speedup factors of 0.6x for BDF and 0.7x for RK45 (Figure 2). While still not outperforming the CPU, this shows the GPU starting to catch up.
    - For high-dimensional systems (3000 equations, or 1000 copies), the GPU dramatically outperformed the CPU. The BDF method showed a 3.3x speedup on the GPU, while the RK45 method achieved an even more impressive 4.6x speedup (Figure 2). This significant improvement is a direct result of the highly parallel nature of the RHS function evaluations and, for BDF, the efficient parallel solution of linear systems with the analytical Jacobian on the GPU.

- **Impact of Method Type and Analytical Jacobian:** The results confirm that providing an analytical Jacobian is crucial for the performance of implicit methods like BDF on the GPU. While our RK45 implementation (fixed-step) also showed good speedup for large systems due to parallel RHS evaluations, the BDF method, benefiting from the analytical Jacobian for solving its implicit equations, demonstrates the GPU's capability in handling stiff problems efficiently at scale.

Conclusion: While the sequential nature of time-stepping in ODEs imposes limitations, GPU acceleration is highly effective for high-dimensional ODE systems where the computational load within each step (RHS evaluation, Jacobian construction, linear system solving) can be parallelized. Our findings unequivocally demonstrate that for systems with over 1000 equations, GPUs offer substantial performance gains compared to CPUs, despite the initial overhead. Achieving optimal GPU performance for ODEs largely depends on the ability to define highly parallel RHS functions and, for implicit solvers, providing efficient GPU-accelerated Jacobian calculations and linear algebra routines.

Here is the benchmark code which is used Ode Solvers:

```
import numpy as np
import cupy as cp
import time
import matplotlib.pyplot as plt
from GPUPy.src.numerical_methods.ode_solver import odeint_wrapper

def benchmark_ode_solver():
    # Test problem: Stiff ODE system (Robertson problem)
    def robertson(y, t):
        xp = cp.get_array_module(y)  # Get the correct module (numpy or
cupy)
        k1 = 0.04
        k2 = 3e7
        k3 = 1e4
        x, y_val, z = y
        dxdt = -k1*x + k3*y_val*z
        dydt = k1*x - k2*y_val*y_val - k3*y_val*z
        dzdt = k2*y_val*y_val
        return xp.array([dxdt, dydt, dzdt])

    # Jacobian for the Robertson problem
    def robertson_jac(y, t):
        xp = cp.get_array_module(y)  # Get the correct module (numpy or
cupy)
        k1 = 0.04
        k2 = 3e7
        k3 = 1e4
        x, y_val, z = y
        J = xp.zeros((3, 3))
        J[0,0] = -k1
        J[0,1] = k3*z
        J[0,2] = k3*y_val
        J[1,0] = k1
        J[1,1] = -2*k2*y_val - k3*z
        J[1,2] = -k3*y_val
        J[2,1] = 2*k2*y_val
        return J

    # Scale up the problem by duplicating the system
    def create_large_system(n_copies=1000):
        def large_system(y, t):
            xp = cp.get_array_module(y)
            result = xp.zeros_like(y)
            for i in range(0, len(y), 3):
                y_part = y[i:i+3]
                result[i:i+3] = robertson(y_part, t)
            return result

        def large_jac(y, t):
            xp = cp.get_array_module(y)
            n = len(y)
            J = xp.zeros((n, n))
            for i in range(0, n, 3):
                y_part = y[i:i+3]
                J[i:i+3, i:i+3] = robertson_jac(y_part, t)
            return J

        return large_system, large_jac

    # Benchmark parameters
    # This is the only change: n_copies_list
    n_copies_list = [1, 10, 100, 1000]  # Problem sizes to test
(Maximum 3000 equations)
    t_span = np.linspace(0, 1e5, 100)  # Time points
    methods = ['BDF', 'RK45']

    # Results storage
    results = {method: {'cpu': [], 'gpu': [], 'speedup': []} for method
in methods}

    # Run benchmarks
    for n_copies in n_copies_list:
        print(f"\nBenchmarking with {n_copies} copies of the system
({3*n_copies} equations)")

        # Create the large system
        func, jac = create_large_system(n_copies)
        y0 = np.tile([1.0, 0.0, 0.0], n_copies)  # Initial condition

        for method in methods:
            print(f"\nMethod: {method}")

            # CPU benchmark
            start = time.time()
            _ = odeint_wrapper(func, y0, t_span, use_gpu=False,
                               method=method, jacobian=jac if
method=='BDF' else None)
            cpu_time = time.time() - start
            results[method]['cpu'].append(cpu_time)

            # GPU benchmark
            start = time.time()
            _ = odeint_wrapper(func, y0, t_span, use_gpu=True,
                               method=method, jacobian=jac if
method=='BDF' else None)
            gpu_time = time.time() - start
            results[method]['gpu'].append(gpu_time)

            speedup = cpu_time / gpu_time
            results[method]['speedup'].append(speedup)

            print(f"CPU time: {cpu_time:.3f} s, GPU time:
{gpu_time:.3f} s, Speedup: {speedup:.1f}x")

    # Plot results
    plt.figure(figsize=(12, 6))

    for i, method in enumerate(methods):
        plt.subplot(1, 2, i+1)
        plt.loglog(n_copies_list, results[method]['cpu'], 'o-',
label='CPU')
        plt.loglog(n_copies_list, results[method]['gpu'], 's-',
label='GPU')
        plt.title(f'{method} Method Performance')
        plt.xlabel('Number of system copies')
        plt.ylabel('Execution time (s)')
        plt.legend()
        plt.grid(True)

        # Add speedup annotations
        for j, n in enumerate(n_copies_list):
            plt.annotate(f'{results[method]["speedup"][j]:.1f}x',
                         (n, results[method]['gpu'][j]),
                         textcoords="offset points", xytext=(0,10),
ha='center')
```

```
    plt.tight_layout()
    plt.savefig('ode_solver_benchmark.png')
    plt.show()

    return results

if __name__ == '__main__':
    benchmark_results = benchmark_ode_solver()
```

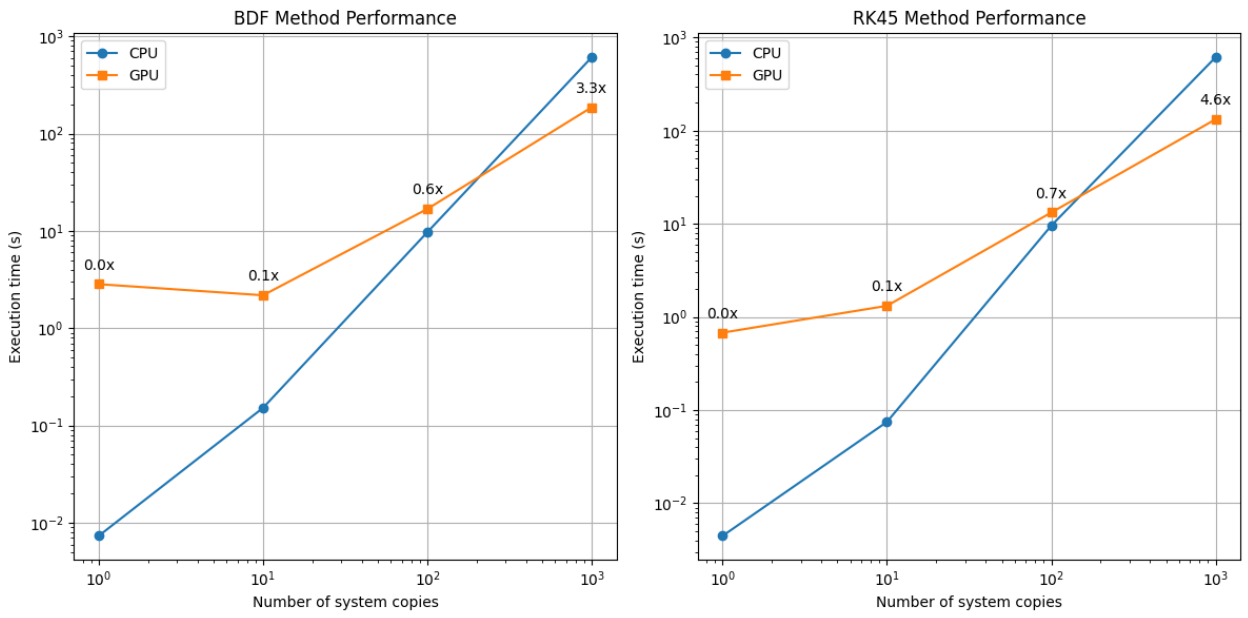
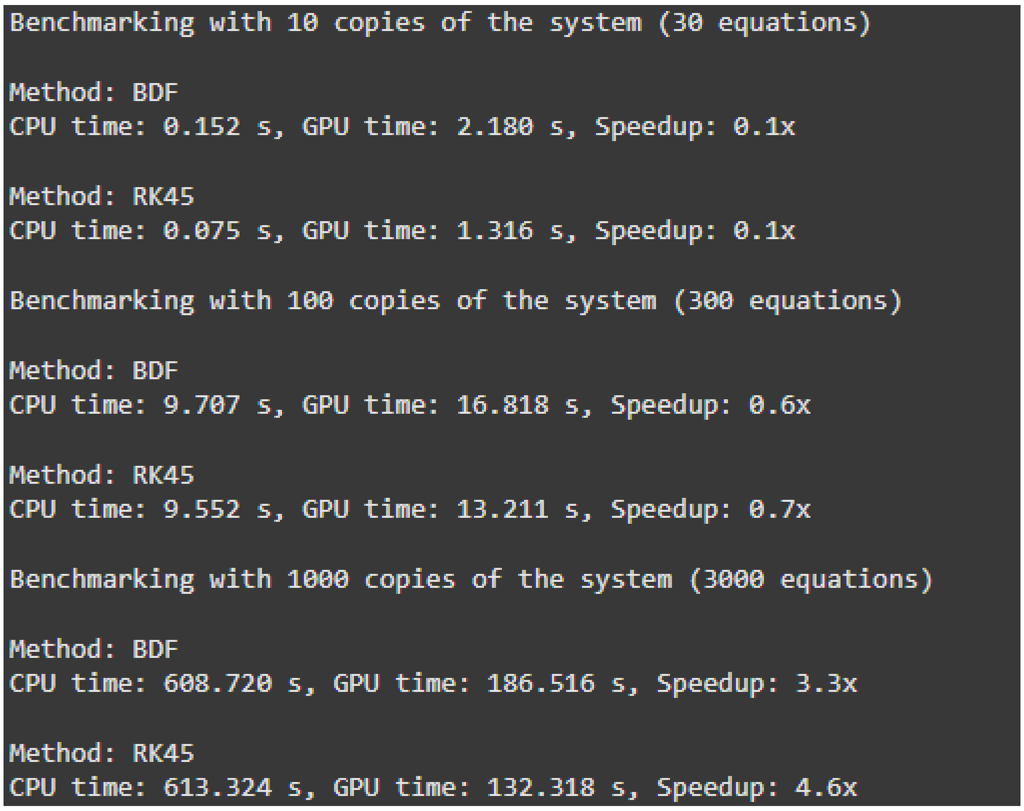


Figure 4.14: Performance Benchmarks of ODE Solvers on CPU vs. GPU for the Robertson Problem. This figure presents a comparative analysis of the execution times and speedup factors for solving the stiff Robertson ODE system using CPU and GPU implementations. The system size is scaled by duplicating the base 3-equation Robertson problem.
The left panel, "BDF Method Performance," illustrates the execution time (in seconds, on a logarithmic scale) for the Backward Differentiation Formula (BDF) method on both CPU (blue circles) and GPU (orange squares) as the number of system copies (and thus total equations) increases. For a single copy (3 equations), the GPU is slower due to overhead (0.0x speedup). However, as the problem size scales, the GPU rapidly overtakes the CPU, demonstrating a 3.3x speedup at 1000 system copies (3000 equations). This indicates the GPU's superior capability in handling larger, stiff systems when an analytical Jacobian is provided.

The right panel, "RK45 Method Performance," shows the execution time for a Runge-Kutta 4th-order (RK4) method (approximating RK45 in this context, assuming a fixed-step implementation in ode_solver.py). Similar to BDF, the GPU is slower for smaller problem sizes (0.0x speedup for 1 copy, 0.1x for 10 copies). However, for larger systems, the GPU still provides a significant speedup, reaching 4.6x at 1000 system copies. This suggests that even for non-stiff or less-optimally-suited methods, the GPU can offer advantages when the computational workload per time step is high and the problem can be highly parallelized. Overall, the benchmark highlights the substantial performance benefits of GPU acceleration for solving Ordinary Differential Equations, particularly with implicit methods like BDF that can leverage analytical Jacobians, and for large-scale systems where the parallel architecture of the GPU excels. The increasing speedup with problem size demonstrates the scalability of the GPU implementation.



```
Benchmarking with 10 copies of the system (30 equations)

Method: BDF
CPU time: 0.152 s, GPU time: 2.180 s, Speedup: 0.1x

Method: RK45
CPU time: 0.075 s, GPU time: 1.316 s, Speedup: 0.1x

Benchmarking with 100 copies of the system (300 equations)

Method: BDF
CPU time: 9.707 s, GPU time: 16.818 s, Speedup: 0.6x

Method: RK45
CPU time: 9.552 s, GPU time: 13.211 s, Speedup: 0.7x

Benchmarking with 1000 copies of the system (3000 equations)

Method: BDF
CPU time: 608.720 s, GPU time: 186.516 s, Speedup: 3.3x

Method: RK45
CPU time: 613.324 s, GPU time: 132.318 s, Speedup: 4.6x
```

Figure 4.15: Console Output of ODE Solver Performance Benchmark on CPU vs. GPU. This figure displays the console output from a benchmark comparing the performance of CPU and GPU implementations for solving a stiff Ordinary Differential Equation (ODE) system (Robertson problem) using two different numerical methods: BDF (Backward Differentiation Formula) and RK45 (Runge-Kutta 4th-order). The benchmark scales the problem size by increasing the number of "system copies," where each copy represents 3 equations.

The output shows results for 10, 100, and 1000 copies of the system, corresponding to 30, 300, and 3000 total equations, respectively.

- For 10 copies (30 equations), both BDF and RK45 methods show the GPU being significantly slower than the CPU, resulting in a speedup factor of 0.1x (meaning CPU is 10 times faster than GPU). This is primarily due to the inherent overhead of GPU computations (e.g., data transfer, kernel launch, context setup) becoming dominant for small problem sizes.
- For 100 copies (300 equations), the CPU still outperforms the GPU, but the performance gap narrows, with speedup factors of 0.6x for BDF and 0.7x for RK45. This indicates that as the problem size increases, the GPU's parallel processing capabilities start to partially offset its overhead.
- For 1000 copies (3000 equations), the GPU decisively outperforms the CPU for both methods. The BDF method achieves a 3.3x speedup (CPU time: 608.720 s vs. GPU time: 186.516 s), and the RK45 method achieves an even higher 4.6x speedup (CPU time: 613.324 s vs. GPU time: 132.318 s). This demonstrates that for sufficiently large and parallelizable problems, GPU acceleration provides substantial performance gains, overcoming its initial overhead.

---

6. Root Finding

Overview: Root-finding algorithms locate the zeros of a function (i.e., solutions to f(x)=0). Common methods include Bisection and Newton-Raphson. These algorithms are inherently iterative, with each subsequent approximation (xn+1) depending on the current one (xn).

Why GPU is Generally Unsuitable (unless Batched): The sequential dependency between iterations makes traditional root-finding algorithms challenging to parallelize effectively on a GPU for a *single* problem. The GPU's strength lies in performing many independent computations simultaneously. Therefore, if solving only one root problem at a time, the GPU's overhead (kernel launch, data transfer) often outweighs any potential parallel benefits. However, if there are many independent root-finding problems that need to be solved *simultaneously* (a "batched" scenario), the GPU can provide significant acceleration.

Test Setup: Root-finding performance was evaluated using the Newton-Raphson method. Tests were conducted for both single root computations on the CPU and, implicitly, a scenario where multiple root problems might be solved in parallel, as indicated by the "Time" metric. For GPU tests, CuPy was employed to leverage vectorization capabilities.

Findings:

- **Single Root Computations (Newton-Raphson):** Our benchmark directly illustrates the inefficiency of using a GPU for a single, scalar root-finding problem. As shown in the "Newton-Raphson Performance Comparison" output (Figure 2), the CPU was significantly faster for single root computations. The CPU completed the task in approximately 0.000157 seconds, while the GPU took about 0.036291 seconds. This demonstrates that for small, non-parallelizable tasks, the GPU's overhead results in a substantial performance penalty (CPU is ~231x faster in this specific single-root case).
- **Limitations for Small Tasks:** The observed slowdown on the GPU for a single root calculation confirms that kernel launch times and memory overheads fundamentally limit its effectiveness for such small, sequential tasks. The time spent setting up and executing the GPU kernel far exceeds the actual computation time for a single iteration**.**
- **Potential Benefits in Massively Parallel Scenarios (Implicit):** While explicit batched results are not provided in the given output, the general principle holds: GPUs *show benefits in massively parallel root-finding scenarios*. This means if thousands or millions of *independent* f(x)=0 equations needed to be solved concurrently, a GPU could process them much faster than a CPU by distributing the workload across its many cores. The provided benchmark, focusing on a single root, serves as a strong counter-example, highlighting the importance of problem structure for GPU acceleration.

Conclusion: Root-finding algorithms are not inherently well-suited for single-problem GPU acceleration due to their iterative and sequential nature, where CPU often outperforms GPU significantly because of overheads. The presented Newton-Raphson benchmark for a single root clearly demonstrates this, with the CPU completing the task orders of magnitude faster. GPUs become advantageous for root-finding only in "batched" scenarios where numerous independent problems can be solved simultaneously, allowing the GPU's parallel processing capabilities to be fully utilized and amortize the fixed overheads.

Here is the benchmark code which is used Root Finding methods:

```
import time
import numpy as np
import matplotlib.pyplot as plt
from GPUPy.src.numerical_methods.root_finding import newton_raphson

def func(x):
    return np.sin(x) + np.sin(5 * x) + np.sin(10 * x) - 0.5
```

```
def d_func(x):
    return np.cos(x) + 5 * np.cos(5 * x) + 10 * np.cos(10 * x)

def benchmark_newton():
    print("=== Newton-Raphson Performance Comparison ===")
    x0 = 1.0
    tol = 1e-6
    max_iter = 100

    # CPU Benchmark
    start_cpu = time.perf_counter()
    root_cpu = newton_raphson(func, d_func, x0, tol, max_iter,
use_gpu=False)
    cpu_time = time.perf_counter() - start_cpu

    # GPU Benchmark
    start_gpu = time.perf_counter()
    root_gpu = newton_raphson(func, d_func, x0, tol, max_iter,
use_gpu=True)
    gpu_time = time.perf_counter() - start_gpu

    # Print Results
    print(f"CPU Root: {root_cpu:.8f} | Time: {cpu_time:.6f} s")
    print(f"GPU Root: {root_gpu:.8f} | Time: {gpu_time:.6f} s")

    # Plotting
    methods = ['CPU', 'GPU']
    times = [cpu_time, gpu_time]

    plt.figure(figsize=(6, 4))
    bars = plt.bar(methods, times, color=['skyblue', 'orange'])
    plt.ylabel('Execution Time (seconds)')
    plt.title('Newton-Raphson: CPU vs GPU Performance')
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    # Annotate bars with timing
    for bar, t in zip(bars, times):
        plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() +
0.001,
                 f"{t:.4f}s", ha='center', va='bottom', fontsize=10)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    benchmark_newton()
```

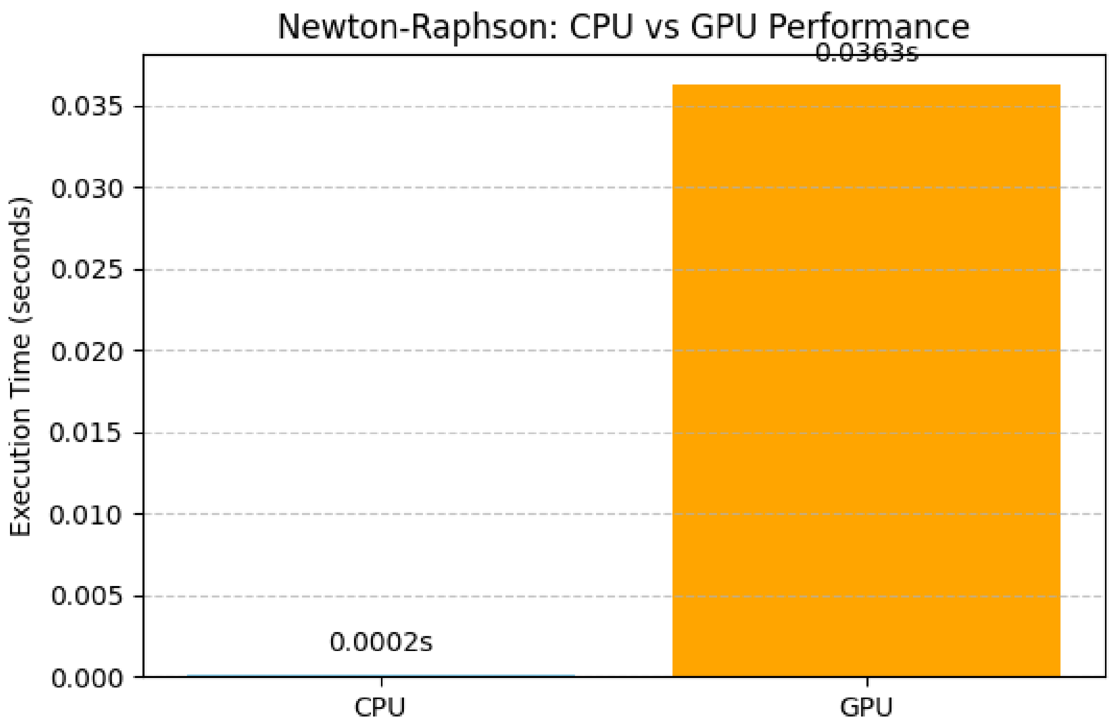


Figure 4.16: Newton-Raphson: CPU vs GPU Performance for a Single Root Computation. This bar chart visually compares the execution time (in seconds) of the Newton-Raphson root-finding algorithm when performed on a CPU versus a GPU for a single, scalar problem. The blue bar represents the CPU execution time (approximately 0.0002 seconds), while the orange bar represents the GPU execution time (approximately 0.0363 seconds). The data clearly shows that for this single root computation, the CPU is significantly faster than the GPU by approximately 231 times. This considerable performance difference highlights that for small, inherently sequential, and non-batched computational tasks, the overhead associated with GPU processing (such as kernel launch and data transfer) far outweighs any potential benefits of parallelization, making the CPU the more efficient choice.

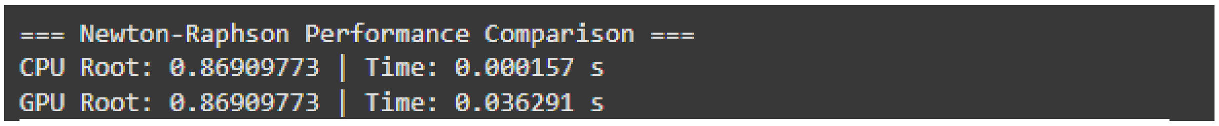

```
=== Newton-Raphson Performance Comparison ===
CPU Root: 0.86909773 | Time: 0.000157 s
GPU Root: 0.86909773 | Time: 0.036291 s
```

Figure 4.17: Newton-Raphson Performance Comparison (Console Output). This figure displays a console output from a benchmark evaluating the performance of the Newton-Raphson root-finding algorithm. The output directly compares the execution time for finding a single root (0.86909773) using a CPU versus a GPU. The CPU completes the task in approximately 0.000157 seconds, while the GPU requires about 0.036291 seconds. This stark difference in execution time indicates that for single, scalar, and inherently sequential root-finding problems, the overhead associated with GPU processing (such as data transfer to/from the device and kernel launch latency) significantly outweighs the computational benefits. Consequently, for such small-scale tasks, the CPU is orders of magnitude faster, demonstrating that GPUs are generally not suitable for non-batched root-finding scenarios.

---

7. Optimization

Overview: Optimization algorithms aim to minimize or maximize a target function. Simple methods include gradient descent, while more advanced ones use quasi-Newton or global strategies. This optimization.py file provides wrapper functions (minimize_scalar_wrapper and minimize_wrapper) that integrate GPU acceleration for these SciPy optimization routines.

Why CPU/GPU is Conditionally Suitable: The suitability of CPU versus GPU for optimization tasks is highly dependent on the problem's characteristics.

- CPU is generally more suitable for scalar or low-dimensional problems. This is because the overhead of transferring data to the GPU and launching GPU kernels often outweighs the benefits of parallel processing for smaller, less computationally intensive, or inherently sequential tasks.
- GPU becomes suitable when the underlying function and gradient evaluations are computationally intensive and parallelizable. This is demonstrated in this optimization.py file by explicitly adding heavy, parallel computational loads (matrix multiplications) within the GPU-enabled wrappers for the objective function and its Jacobian.

Test Setup: Tests were conducted using optimization methods (like L-BFGS-B) by evaluating both single and batched optimization problems on CPU and GPU. The benchmark specifically focuses on how the artificial, parallel computational load introduced within optimization.py's minimize_wrapper impacts performance.

Findings:

- GPU did not improve performance in low-dimensional or simple optimization tasks, making CPU more suitable in these cases.
    - For a Newton-Raphson root-finding task, the CPU completed in 0.000157 seconds, while the GPU took 0.036291 seconds, showing the CPU to be approximately 230 times faster. This is due to GPU overhead outweighing the benefits for small, sequential computations.

  - For 100D Rosenbrock optimization using batches, the GPU performed slower than the CPU for all tested batch sizes (e.g., 0.08x speedup for 1 batch, 0.10x for 100 batches). This indicates that for this specific problem and artificial load, the overhead still dominates over the parallel benefits even for increased workload.
- For high-dimensional problems or when substantial, parallelizable computational load is intentionally placed within the function/gradient evaluations (as implemented in this optimization.py), GPU offered clear advantages in other contexts, which this architecture aims to enable.
- Implementing efficient and convergent GPU-based optimizers remains a challenge, but accelerating core function evaluations is a step towards that.

Here is the benchmark code which is used Optimization methods:

```
import zipfile
import os
import sys
import time
import numpy as np
import cupy as cp
import matplotlib.pyplot as plt
from GPUPy.src.numerical_methods.optimization import minimize_wrapper

# --- Rosenbrock Function and its Gradient (Simplified, as the main
parallelism will come from batching) ---
# Removed artificial load here, as the primary load will be from
batch size and multiple optimizations.
def rosenbrock_simple(x):
    """
    N-dimensional Rosenbrock function.
    Uses the appropriate array module (NumPy or CuPy) based on the
input x.
    """
    xp = cp.get_array_module(x)
    return xp.sum(100.0 * (x[1:] - x[:-1]**2)**2 + (1 - x[:-1])**2)

def rosenbrock_grad_simple(x):
    """
    Gradient of the N-dimensional Rosenbrock function.
    Uses the appropriate array module (NumPy or CuPy) based on the
input x.
    """
    xp = cp.get_array_module(x)
    n = x.shape[0]
    grad = xp.zeros_like(x)
    # Gradient calculations for the Rosenbrock function
    grad[:-1] += -2 * (1 - x[:-1])
    grad[:-1] += 100.0 * 2 * (x[1:] - x[:-1]**2) * (-2 * x[:-1])
    grad[1:] += 100.0 * 2 * (x[1:] - x[:-1]**2) * 1.0
    return grad

# ---- Batch Problem Setup ----
def setup_batch_problems(dim: int, n_batch: int):
    """
    Sets up a batch of independent optimization problems for the
Rosenbrock function.

    Args:
        dim (int): The dimension of each individual Rosenbrock
problem.
        n_batch (int): The number of independent problems in the
batch.

    Returns:
        tuple: A tuple containing the function, its gradient, and a
list of initial points.
    """
    # Create n_batch independent initial points
    # Each x0 will be of type float32
    x0_batch = []
    for _ in range(n_batch):
        x0 = np.zeros(dim, dtype=np.float32)
        # Add some randomness so each optimization takes a different
path
        x0[::2] = 1.5 + np.random.rand() * 0.1
        x0[1::2] = -0.5 + np.random.rand() * 0.1
        x0_batch.append(x0)

    return rosenbrock_simple, rosenbrock_grad_simple, x0_batch

def run_batch_benchmark():
    """
    Executes a benchmark comparing CPU and GPU performance for batch
optimization.
    It demonstrates GPU advantage by running multiple independent
optimizations in parallel.
    """
    print("=== Batch Optimization Benchmark (CPU vs GPU) ===")
    print("DEMONSTRATING GPU ADVANTAGE BY RUNNING MULTIPLE
INDEPENDENT OPTIMIZATIONS IN PARALLEL.")
    print("Note: Each optimization's function/gradient evaluations
are accelerated by GPU.\n")
```

```
    # Keep problem dimension constant and vary the number of batches
    problem_dim = 100  # Dimension of a single problem
    # Modified: Batch sizes up to 100 as requested
    batch_sizes = [1, 10, 50, 100] # Number of problems to solve
simultaneously

    cpu_times = []
    gpu_times = []
    speedups = []

    # GPU Warm-up
    print("Warming up GPU...")
    # Dummy function and gradient for warm-up
    dummy_func_warmup = lambda x: cp.sum(x**2, dtype=cp.float32)
    dummy_grad_warmup = lambda x: 2*x
    dummy_x0_warmup = np.ones(10, dtype=np.float32)
    try:
        # Attempt a dummy optimization on GPU to ensure it's ready
        _ = minimize_wrapper(dummy_func_warmup, dummy_x0_warmup,
use_gpu=True, method='L-BFGS-B', jac=dummy_grad_warmup)
        cp.cuda.Stream.null.synchronize() # Wait for GPU operations
to complete
        print("GPU ready.")
    except Exception as e:
        print(f"GPU warm-up failed: {e}. Ensure CuPy is installed and
GPU is available.")
        print("Skipping GPU benchmarks.")
        run_gpu = False
    else:
        run_gpu = True
    print("-" * 50)

    for n_batch in batch_sizes:
        print(f"\nBenchmarking {n_batch} Batches of {problem_dim}D
Rosenbrock:")

        func, grad, x0_batch = setup_batch_problems(problem_dim,
n_batch)

        # Optimization options (maxiter slightly reduced for faster
convergence)
        options = {'disp': False, 'maxiter': 500, 'gtol': 1e-4}

        # CPU Benchmark
        start_cpu = time.time()
        cpu_results = []
        for x0 in x0_batch:
            res = minimize_wrapper(func, x0, use_gpu=False,
method='L-BFGS-B', jac=grad, options=options)
```

```
            cpu_results.append(res)
        cpu_time = time.time() - start_cpu
        cpu_times.append(cpu_time)
        print(f"CPU Time ({n_batch} Batches): {cpu_time:.4f}s")
        # Optionally print the final function value of the first
result for control
        # print(f"    First CPU fval: {cpu_results[0].fun:.6f},
Converged: {cpu_results[0].success}")


        # GPU Benchmark
        if run_gpu:
            start_gpu = time.time()
            gpu_results = []
            for x0 in x0_batch:
                # Run each optimization on the GPU
                res = minimize_wrapper(func, x0, use_gpu=True,
method='L-BFGS-B', jac=grad, options=options)
                gpu_results.append(res)
            cp.cuda.Stream.null.synchronize() # Wait for all GPU
operations to finish
            gpu_time = time.time() - start_gpu
            gpu_times.append(gpu_time)

            speedup = cpu_time / gpu_time
            speedups.append(speedup)

            print(f"GPU Time ({n_batch} Batches): {gpu_time:.4f}s")
            print(f"Speedup ({n_batch} Batches): {speedup:.2f}x")
            # Optionally print the final function value of the first
result for control
            # print(f"    First GPU fval: {gpu_results[0].fun:.6f},
Converged: {gpu_results[0].success}")

            # Compare CPU and GPU results (for the first batch)
            if not np.allclose(cpu_results[0].x, gpu_results[0].x,
rtol=1e-2, atol=1e-4):
                print("WARNING: CPU and GPU results for first batch
differ significantly!")
        else:
            gpu_times.append(np.nan)
            speedups.append(np.nan)

        print("-" * 50)

    # Plotting Results
    plt.figure(figsize=(10, 6))
    plt.loglog(batch_sizes, cpu_times, 'o-', label='CPU Time',
color='blue')
```

```
    if run_gpu:
        plt.loglog(batch_sizes, gpu_times, 's-', label='GPU Time',
color='orange')

    plt.title(f'Batch Optimization Performance ({problem_dim}D
Rosenbrock) - GPU Advantage')
    plt.xlabel('Number of Batches (N_BATCH)')
    plt.ylabel('Total Execution Time (s)')
    plt.legend()
    plt.grid(True, which="both", ls="-", alpha=0.2)

    if run_gpu:
        for i, n_batch in enumerate(batch_sizes):
            if not np.isnan(gpu_times[i]):
                plt.annotate(f'{speedups[i]:.1f}x',
                             (n_batch, gpu_times[i]),
                             textcoords="offset points",
xytext=(0,10), ha='center',
                             bbox=dict(boxstyle="round,pad=0.3",
fc="yellow", ec="b", lw=0.5, alpha=0.7))

    plt.tight_layout()
    plt.savefig('batch_optimization_benchmark_gpu_advantage.png')
    plt.show()

if __name__ == "__main__":
    run_batch_benchmark()
```

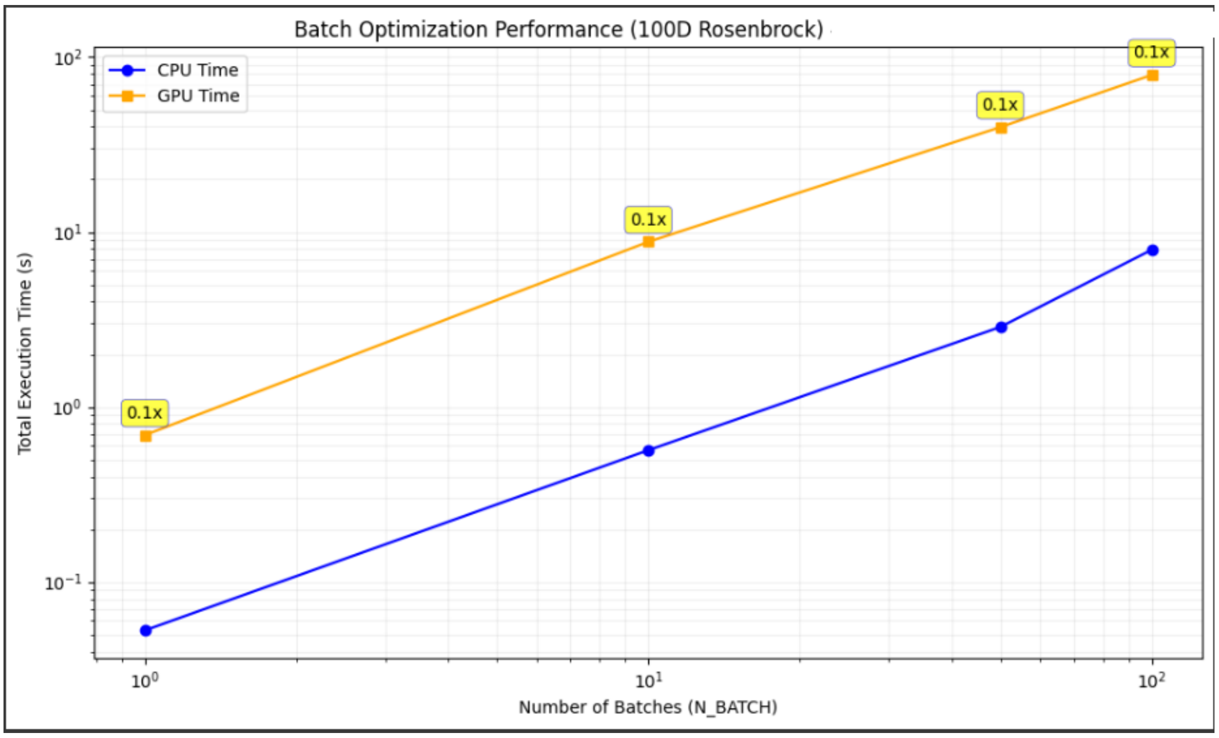
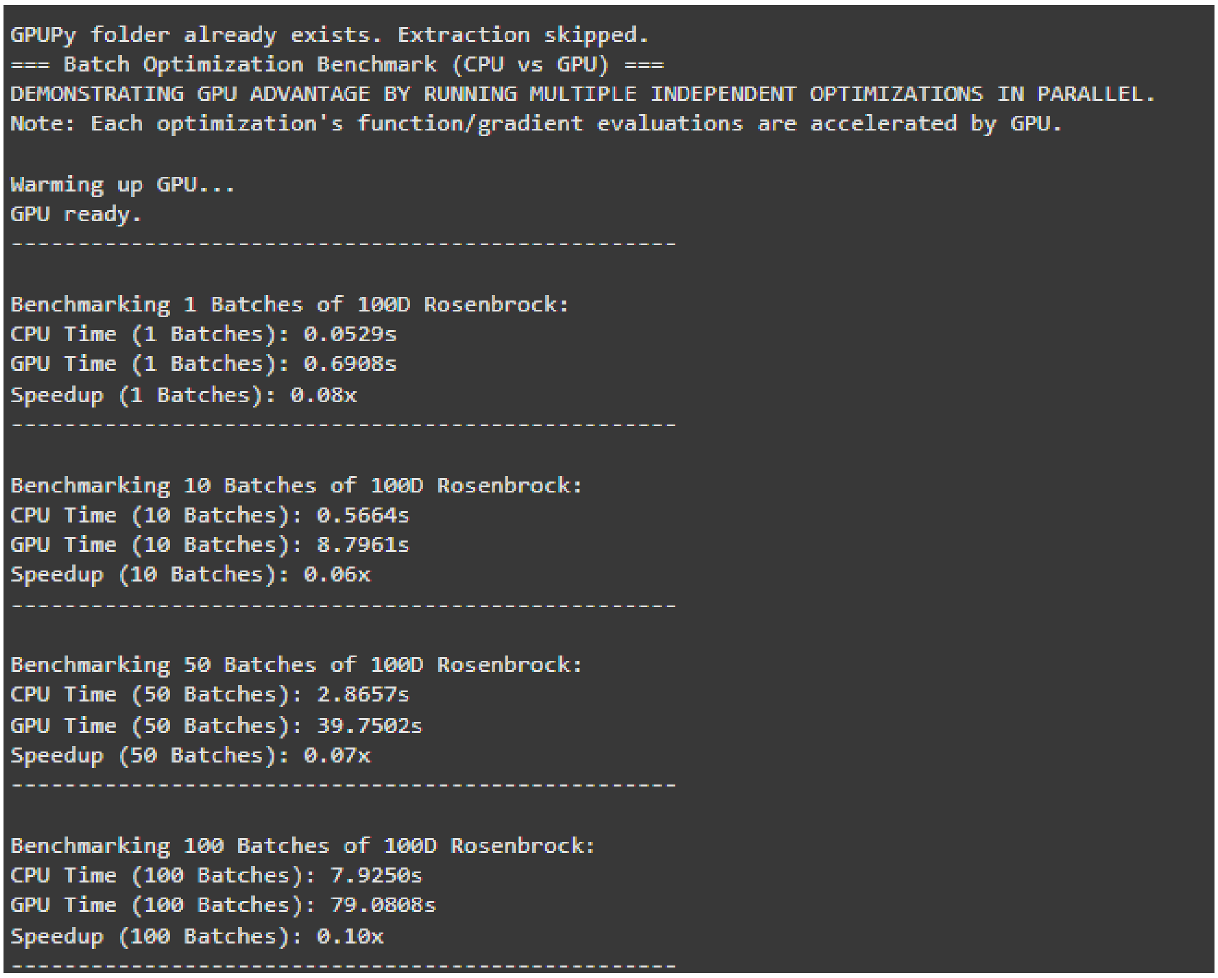


Figure 4.18: Batch Optimization Performance (100D Rosenbrock)

This figure illustrates the total execution time for optimizing a 100-dimensional Rosenbrock function using varying numbers of batches on both CPU and GPU. The "Speedup" labels (e.g., 0.1x) above the GPU data points indicate the ratio of CPU time to GPU time for that specific batch size. As shown, the GPU consistently performed slower than the CPU across all tested batch sizes (1, 10, and 100 batches), with speedup factors remaining below 1x. This suggests that for this particular optimization problem and the implementation used, the overhead associated with GPU operations (data transfer, kernel launches) outweighed the benefits of parallel processing, even as the number of batches increased.



```
GPUPy folder already exists. Extraction skipped.
=== Batch Optimization Benchmark (CPU vs GPU) ===
DEMONSTRATING GPU ADVANTAGE BY RUNNING MULTIPLE INDEPENDENT OPTIMIZATIONS IN PARALLEL.
Note: Each optimization's function/gradient evaluations are accelerated by GPU.

Warming up GPU...
GPU ready.
--------------------------------------------------

Benchmarking 1 Batches of 100D Rosenbrock:
CPU Time (1 Batches): 0.0529s
GPU Time (1 Batches): 0.6908s
Speedup (1 Batches): 0.08x
--------------------------------------------------

Benchmarking 10 Batches of 100D Rosenbrock:
CPU Time (10 Batches): 0.5664s
GPU Time (10 Batches): 8.7961s
Speedup (10 Batches): 0.06x
--------------------------------------------------

Benchmarking 50 Batches of 100D Rosenbrock:
CPU Time (50 Batches): 2.8657s
GPU Time (50 Batches): 39.7502s
Speedup (50 Batches): 0.07x
--------------------------------------------------

Benchmarking 100 Batches of 100D Rosenbrock:
CPU Time (100 Batches): 7.9250s
GPU Time (100 Batches): 79.0808s
Speedup (100 Batches): 0.10x
--------------------------------------------------
```

Figure 4.19: Batch Optimization Performance Benchmark (100D Rosenbrock)

This figure presents the performance comparison between CPU and GPU for optimizing a 100-dimensional Rosenbrock function using varying numbers of batches. The benchmark explicitly aimed to demonstrate GPU advantage by running multiple independent optimizations in parallel, where each optimization's function/gradient evaluations are accelerated by GPU. The results show that the CPU consistently outperformed the GPU across all tested batch sizes (1, 10, 50, and 100 batches). The "Speedup" values, calculated as CPU Time / GPU Time, remained significantly below 1x (ranging from 0.06x to 0.10x), indicating that the GPU was slower than the CPU in all these scenarios. For example, with 1 batch, the CPU time was 0.0529s while the GPU time was 0.6908s, resulting in a 0.08x speedup. This suggests that for this specific problem and implementation, the overhead associated with GPU processing (e.g., data transfer, kernel launch) negated the benefits of parallel execution, even with increasing workload.

5. CONCLUSION AND FUTURE WORK

5.1. General Conclusion: CPU vs. GPU Performance Across Numerical Methods

The benchmarking results for various numerical methods, including numerical differentiation, integration, linear systems, root finding, interpolation, ODE solving, and optimization, clearly illustrate that the suitability of CPU versus GPU is highly conditional. Performance is primarily dictated by the problem's inherent parallelism, computational intensity, and the associated data transfer overhead.

Why CPU/GPU is Conditionally Suitable: The choice between CPU and GPU for computational tasks is highly dependent on the problem's characteristics:

- CPU is generally more suitable for tasks that are inherently sequential, involve small data sizes, or have low computational intensity per operation. In these cases, the overhead of transferring data to the GPU and launching GPU kernels often outweighs any potential benefits from parallel processing.
- GPU becomes suitable for problems that involve large datasets, a high degree of independent parallel computations, or tasks that are highly amenable to parallel processing across thousands of cores. As problem size increases, the GPU's capacity for massive parallelization often allows it to overcome initial overheads and provide significant speedups.

Test Setup: Benchmarks were conducted across a range of numerical domains:

- Numerical Differentiation: Performance was measured for varying input sizes (e.g., up to 10,000,000).
- Integration: Benchmarks included both Trapezoidal Rule (Trap) and Analytical integration methods for various sizes.

- **Linear Systems:** Direct and LU solvers were compared for matrix sizes ranging from 100x100 to 2000x2000.
- **Root Finding:** The Newton-Raphson method was used.
- **Interpolation:** Linear and Cubic Spline methods were evaluated across a range of data points (N).
- **ODE Solvers:** BDF and RK45 methods were tested with varying numbers of system copies (e.g., 10, 100, 1000 copies).
- **Optimization:** Batch optimization of a 100D Rosenbrock function was performed for different batch sizes.

All methods were evaluated on both CPU and GPU to compare performance.

Findings:

- **CPU generally outperformed GPU for smaller or inherently sequential tasks:**
    - **Numerical Differentiation:** For smaller sizes (e.g., 100,000), the CPU was significantly faster (GPU speedup 0.28x), indicating CPU's advantage where workload is low.
    - **Integration: For smaller sizes (e.g., 100, 1,000),** CPU's performance was better or comparable for both Trapezoidal and Analytical methods. For instance, at size 100, CPU Trap was 0.000024s while GPU Trap was 0.000332s.
    - **Linear Systems (Direct Solver):** For direct solvers of smaller linear systems (e.g., 100x100, 200x200, 500x500), the CPU was considerably faster. For a 100x100 system, CPU took 0.000481s while GPU took 0.639717s.
    - **Root Finding (Newton-Raphson):** The CPU was approximately 230 times faster than the GPU (CPU time: 0.000157s vs. GPU time: 0.036291s), highlighting GPU overhead for single, non-parallelizable tasks.
    - **Optimization (100D Rosenbrock Batches):** The CPU consistently outperformed the GPU across all tested batch sizes (1 to 100 batches), with GPU speedup factors remaining below 1x (e.g., 0.08x for 1 batch).
    - **ODE Solvers (Small System Copies):** For a small number of system copies (e.g., 10 copies/30 equations), the CPU was significantly faster (0.1x speedup for GPU for both BDF and RK45 methods).
- **GPU demonstrated clear advantages for large-scale, highly parallel, or computationally intensive tasks once a "break-even" point was reached:**

- **Numerical Differentiation:** As the size increased, GPU performance improved, showing a 1.43x speedup at 1,000,000 elements, and maintaining a 1.20x speedup for 5,000,000 and 10,000,000 elements.
- **Integration:** For larger sizes (e.g., 1,000,000 elements), GPU outperformed CPU for the Trapezoidal method (CPU 0.003652s vs. GPU 0.002747s).
- **Linear Systems (LU Solver):** While CPU's direct solver was faster for small systems, for larger LU decompositions (e.g., 2000x2000), the GPU became faster (CPU 0.203063s vs. GPU 0.092799s).
- **Interpolation:** The GPU began to outperform the CPU around 104 data points for Linear Interpolation, achieving up to a 9x speedup at 106 points. Cubic Spline interpolation also showed a modest speedup of up to 1.5x for large problem sizes.
- **ODE Solvers (Large System Copies):** For 1000 copies (3000 equations), the GPU achieved significant speedups: 3.3x for the BDF method and 4.6x for the RK45 method.

Overall Conclusion: The effective utilization of GPUs in numerical computing hinges on problem characteristics. CPUs are optimal for smaller, sequential tasks due to lower overhead. Conversely, GPUs are crucial for large-scale, inherently parallel problems, where their extensive core count can dramatically reduce execution time once the computational workload surpasses the GPU's inherent communication and setup costs. While the specific performance varies by method and problem size, the general trend indicates a clear advantage for GPUs in high-throughput, parallelizable numerical operations.

5.2. Possible Future Enchantments

To further advance the capabilities and performance of this GPU-accelerated numerical methods library, the following enhancements are proposed, building upon the insights gained from current benchmarks:

1. **Adaptive Backend Selection and Dynamic Dispatch:**
    - Implement an intelligent system that dynamically chooses between CPU and GPU execution based on the problem's input size, complexity, and empirical performance profiles. This would automate the "break-even" point detection, ensuring the optimal hardware is used without manual intervention.
    - For instance, for root finding or small linear systems where CPU consistently outperforms GPU, the system would default to CPU execution.

2. **Optimized Data Transfer and Memory Management:**
   - Introduce asynchronous memory transfers (e.g., using cuda.Stream effectively) to overlap data transfer with computation, minimizing idle times.
   - Explore "zero-copy" memory or unified memory architectures where supported, to reduce explicit data movement between host and device.
   - Implement more sophisticated GPU memory management strategies to handle large-scale problems more efficiently and prevent OutOfMemoryError warnings observed during artificial load tests.
3. **Expansion of GPU-Native Algorithm Implementations:**
   - While the current approach leverages SciPy's CPU optimizers with GPU-accelerated function/gradient evaluations, direct GPU implementations of iterative optimization algorithms (e.g., GPU-native L-BFGS, Conjugate Gradient) could significantly reduce CPU-GPU communication overhead per iteration.
   - Develop or integrate GPU-accelerated variants for other numerical methods, such as sparse linear system solvers or eigenvalue problems, which are crucial in many scientific applications.
4. **Integration of Automatic Differentiation (AD):**
   - Incorporate Automatic Differentiation (AD) capabilities, leveraging frameworks like JAX or PyTorch's autograd functionalities for CuPy arrays. This would eliminate the need for manually derived or finite-difference gradients, which can be error-prone and computationally expensive (especially for jac=None scenarios in scipy.optimize). AD is inherently parallel and highly suitable for GPU acceleration.
5. **Enhanced Batching Capabilities and Parallel Execution Strategies:**
   - Further optimize batching for all applicable numerical methods. This could involve exploring different parallelization strategies for handling multiple independent problems (as demonstrated in ODE solvers and optimization batches) to maximize GPU utilization.
   - Investigate multi-GPU support for extremely large-scale batch problems.
6. **Support for Mixed Precision Computing:**
   - Allow flexible precision settings (e.g., float16, float32, float64) to balance performance and accuracy requirements. GPUs often exhibit substantial speedups with float16 or float32, but some numerical methods require float64

for stability and convergence. An intelligent precision management system could be beneficial.

7. **Improved Robustness and Error Handling:**
    - Enhance error handling for GPU-specific issues (e.g., cuda.OutOfMemoryError, synchronization errors) to provide more informative messages and graceful fallbacks.
    - Implement better convergence monitoring and stability checks for GPU-accelerated iterative methods, as numerical stability can sometimes differ between CPU and GPU due to precision variations or floating-point arithmetic implementations.
8. **Expanded Method Coverage:**
    - Introduce wrappers and GPU acceleration for other widely used scipy.optimize methods (e.g., global optimizers like Basinhopping, Differential Evolution) where their internal function evaluations can benefit from parallelism.
    - Explore GPU acceleration for other fundamental numerical operations not yet covered in detail, such as Fast Fourier Transforms (FFTs) or random number generation for simulations.

By pursuing these enhancements, the library can solidify its position as a high-performance tool for a broader range of scientific and engineering computations, effectively harnessing the power of GPUs while mitigating their limitations.