Summation by Parts Operators for PDEs

Assignment 2

Abdukhomid Nurmatov

KAUST

27-02-2025

Contents

Problem 1

For a given function $f(x)$ on the interval $[0, L]$, find its derivative at the collocation points using the differentiation matrix D from Problem 1 in Assignment 1.

Solution:

So, given a function $f : [0, L] \to \mathbb{R}$, we need to approximate its derivative $f'(y)$ using the Legendre-Gauss-Lobatto (LGL) differentiation matrix D originally defined on $[-1, 1]$. The main steps are the following:

1. **Affine Transformation of Nodes** The LGL nodes $\{x_i\}_{i=0}^{N}$ on $[-1, 1]$ are mapped to $[0, L]$ via:

$$y_i = \frac{L}{2}(x_i + 1), \quad i = 0, 1, \ldots, N.$$

   This ensures $y_0 = 0$, $y_N = L$, and interior nodes $y_1, \ldots, y_{N-1}$ lie in $(0, L)$.

2. **Scaling the Differentiation Matrix** The derivative relationship between the intervals is governed by the chain rule:

$$\frac{\mathrm{d}f}{\mathrm{d}y} = \frac{\mathrm{d}f}{\mathrm{d}x} \cdot \frac{\mathrm{d}x}{\mathrm{d}y} = \frac{2}{L} \cdot \frac{\mathrm{d}f}{\mathrm{d}x}.$$

   Thus, the differentiation matrix D for $[0, L]$ is scaled by $\frac{2}{L}$:

$$D_{\text{scaled}} = \frac{2}{L} \cdot D.$$

3. **Approximating the Derivative** Given function values $\mathbf{f} = [f(y_0), f(y_1), \ldots, f(y_N)]^T$, the approximate derivative at the nodes is:

$$\mathbf{f}' = D_{\text{scaled}} \cdot \mathbf{f}.$$

With everything stated above and using the code from Problem 1 in Assignment 1, the suggested code is as follows:

```python
import numpy as np
import math

class LGL:
    """
    Class for computing Legendre-Gauss-Lobatto (LGL) nodes, weights, and
    the differentiation matrix D based on the Lagrange basis polynomials.

    Attributes:
        p (int): Degree of the Legendre polynomial (the quadrature has p+1 nodes).
        L (float): Length of the interval [0, L].
        nodes (np.ndarray): The computed LGL nodes on the interval [-1, 1].
        weights (np.ndarray): The computed quadrature weights.
        D (np.ndarray): The differentiation matrix for computing derivatives.
    """

    def __init__(self, p, L=1):
        """
```

```
        Initialize the LGL object with a given polynomial degree p and interval length L.

        Parameters:
            p (int): Degree of the Legendre polynomial.
            L (float, optional): Length of the interval [0, L]. Default is 1.
        """
        self.p = p
        self.L = L
        self.nodes, self.weights = self._compute_nodes_weights()
        self.D = self.differentiation_matrix()

    @staticmethod
    def legendre_poly_coeffs(p):
        """
        Compute the coefficients of the Legendre polynomial P_p(x).

        Parameters:
            p (int): Degree of the Legendre polynomial.

        Returns:
            np.ndarray: Array of coefficients in descending order (highest power first).
        """
        poly_dict = {}
        for k in range(p // 2 + 1):
            power = p - 2 * k
            coeff = ((-1) ** k * math.comb(p, k) * math.comb(2 * p - 2 * k, p)) / (2 ** p)
            poly_dict[power] = coeff
        coeffs = [poly_dict.get(power, 0) for power in range(p, -1, -1)]
        return np.array(coeffs)

    @classmethod
    def legendre_poly(cls, p):
        """
        Construct a numpy.poly1d object representing the Legendre polynomial P_p(x).

        Parameters:
            p (int): Degree of the Legendre polynomial.

        Returns:
            np.poly1d: The Legendre polynomial.
        """
        coeffs = cls.legendre_poly_coeffs(p)
        return np.poly1d(coeffs)

    def _compute_nodes_weights(self):
        """
        Compute the LGL nodes and quadrature weights.

        Returns:
            tuple: (nodes, weights) where
```

```python
                    nodes is a numpy array of the LGL nodes, and
                    weights is a numpy array of the corresponding quadrature weights.
        """
        P = self.legendre_poly(self.p)
        dP = P.deriv()
        interior_nodes = np.sort(dP.r.real)  # Zeros of P'_p(x)
        nodes = np.concatenate(([-1.0], interior_nodes, [1.0]))  # Include endpoints
        weights = 2 / (self.p * (self.p + 1) * (P(nodes) ** 2))
        return nodes, weights

    def differentiation_matrix(self):
        """
        Compute the differentiation matrix D.

        Returns:
            np.ndarray: The differentiation matrix D of shape (N, N), where N = p+1.
        """
        x = self.nodes
        N = len(x)
        D = np.zeros((N, N))
        b = np.zeros(N)  # Barycentric weights
        for j in range(N):
            b[j] = 1.0 / np.prod(x[j] - np.delete(x, j))
        for i in range(N):
            for j in range(N):
                if i != j:
                    D[i, j] = (b[j] / b[i]) / (x[i] - x[j])
            D[i, i] = -np.sum(D[i, :])  # Ensure row sum is zero
        return (2 / self.L) * D  # Rescale for [0, L]

    def transform_nodes(self):
        """
        Transform LGL nodes from [-1, 1] to [0, L].

        Returns:
            np.ndarray: Transformed nodes.
        """
        return (self.L / 2) * (self.nodes + 1)

    def compute_derivative(self, f):
        """
        Compute the derivative of a given function f at the collocation points.

        Parameters:
            f (callable): Function to differentiate.

        Returns:
            tuple: (transformed nodes, derivative values at those nodes).
        """
        x_mapped = self.transform_nodes()
        f_values = f(x_mapped)
        return x_mapped, self.D @ f_values

```

```
def main():
    p = 4  # Degree of Legendre polynomial
    L = 10  # Interval length

    # Define function f(x)
    def f(x):
        return np.full_like(x, 5.0 * x)  # Ensures valid differentiation

    lgl = LGL(p, L)
    y, f_prime = lgl.compute_derivative(f)

    print("Mapped nodes y:")
    print(y)

    print("\nFunction values f(y):")
    print(f(y))

    print("\nComputed derivative df/dy:")
    print(f_prime)

if __name__ == '__main__':
    main()
```

by running which we get the following output:

```
Mapped nodes y:
[ 0.          1.72673165  5.          8.27326835 10.        ]

Function values f(y):
[ 0.          8.63365823 25.         41.36634177 50.        ]

Computed derivative df/dy:
[5. 5. 5. 5. 5.]
```

Or, by changing the function to a constant, we obtain the following output:

```
Mapped nodes y:
[ 0.          1.72673165  5.          8.27326835 10.        ]

Function values f(y):
[5. 5. 5. 5. 5.]

Computed derivative df/dy:
[ 5.13478149e-16  1.87350135e-16 -1.11022302e-16 -1.94289029e-16
 -6.66133815e-16]
```

GitHub

GitHub link for the codes in the folder `Assignment 2`:
`https://github.com/nurmaton/SBP_KAUST/tree/main/Assignment%202`