Summation by Parts Operators for PDEs

Assignment 1

Abdukhomid Nurmatov

KAUST

11-02-2025

Contents

Problem 1 1
Problem 2 8
GitHub 14

Problem 1

Code with Python the entries of the differentiation matrix D.

Solution:

First of all, recall that the Legendre–Gauss–Lobatto (LGL) points on the interval $[-1, 1]$ include the two endpoints:

$$x_0 = -1 \quad \text{and} \quad x_N = 1.$$

The remaining $N-1$ nodes,

$$x_1,\ x_2,\ \ldots,\ x_{N-1},$$

are given by the zeros of the derivative of the Legendre polynomial $P_p(x)$ of degree

$$p = N - 1,$$

where

$$P_p(x) = \frac{1}{2^p} \sum_{k=0}^{\lfloor p/2 \rfloor} (-1)^k \binom{p}{k} \binom{2p-2k}{p} x^{p-2k}.$$

That is, the interior nodes satisfy

$$P_p'(x_i) = 0, \quad i = 1, 2, \ldots, N-1.$$

The weights corresponding to these nodes are given by the formula

$$w_i = \frac{2}{p(p+1)[P_p(x_i)]^2}, \quad i = 0, 1, \ldots, N.$$

In particular, the weights at the endpoints are

$$w_0 = w_N = \frac{2}{p(p+1)}.$$

If we assume that we are given N distinct collocation points (in our case, the LGL points)

$x_1,\, x_2,\, \ldots,\, x_N$ (here, for simplicity of notation, we start indexing from 1),

then we define the Lagrange basis functions as

$$L_j(x) = \prod_{\substack{k=1 \\ k \neq j}}^{N} \frac{x - x_k}{x_j - x_k}, \quad j = 1, \ldots, N.$$

Recall that these functions satisfy

$$L_j(x_i) = \delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

If we interpolate a smooth function $f(x)$ at the points x_i by

$$f_N(x) = \sum_{j=1}^{N} f(x_j)\, L_j(x),$$

then its derivative is

$$f_N'(x) = \sum_{j=1}^{N} f(x_j)\, L_j'(x).$$

Evaluating at the collocation points $x = x_i$ yields

$$f_N'(x_i) = \sum_{j=1}^{N} L_j'(x_i)\, f(x_j).$$

Thus, if we define the entries of the differentiation matrix D by

$$D_{ij} = L_j'(x_i),$$

then applying D to the vector of function values $\big[f(x_1), f(x_2), \ldots, f(x_N)\big]^T$ gives an approximation to the derivative at the nodes.

Before proceeding with the coding, let us derive formulas for the entries of the differentiation matrix D. The claim is that the entries of D are given by:

- For $i \neq j$:

$$D_{ij} = \frac{b_j}{b_i}\,\frac{1}{x_i - x_j}, \text{ with } b_j = \frac{1}{\displaystyle\prod_{\substack{k=1\\k\neq j}}^{N}(x_j - x_k)}.$$

- For $i = j$:

$$D_{ii} = -\sum_{\substack{j=1\\j\neq i}}^{N} D_{ij}.$$

Proof:
☐ The proof is divided into the following two subparts:

1. We claim that

$$D_{ij} = \frac{b_j}{b_i}\,\frac{1}{x_i - x_j},\ i \neq j,\ \text{where } b_i = \frac{1}{\prod_{\substack{k=1\\k\neq i}}^{N}(x_i - x_k)}. \tag{1.1}$$

   Proof:
   ☐ Assume that $i \neq j$. When $x = x_i$ and $i \neq j$, we know that $L_j(x_i) = 0$ (since the basis functions act like "delta–functions" at the nodes). Moreover, since the zero is simple (each $L_j(x)$ has a simple zero at $x = x_i$ when $i \neq j$), we can compute the derivative by "factoring out" the zero. For $i \neq j$, the product

$$L_j(x) = \prod_{\substack{k=1\\k\neq j}}^{N} \frac{x - x_k}{x_j - x_k}$$

   contains a factor corresponding to $k = i$. We can write this factor separately as

$$L_j(x) = \frac{x - x_i}{x_j - x_i} \prod_{\substack{k=1\\k\neq i,j}}^{N} \frac{x - x_k}{x_j - x_k}.$$

Since $L_j(x_i) = 0$, the derivative at $x = x_i$ is given by

$$L_j'(x_i) = \lim_{x \to x_i} \frac{L_j(x) - 0}{x - x_i} = \frac{1}{x_j - x_i} \prod_{\substack{k=1 \\ k \neq i,j}}^{N} \frac{x_i - x_k}{x_j - x_k}.$$

Now, define the barycentric weights as

$$b_j = \frac{1}{\prod\limits_{\substack{k=1 \\ k \neq j}}^{N} (x_j - x_k)}.$$

Next, we split the full product for x_i into two parts:

$$\prod_{\substack{k=1 \\ k \neq i}}^{N} (x_i - x_k) = (x_i - x_j) \prod_{\substack{k=1 \\ k \neq i,j}}^{N} (x_i - x_k) \quad \Longrightarrow \quad \prod_{\substack{k=1 \\ k \neq i,j}}^{N} (x_i - x_k) = \frac{1}{b_i} \frac{1}{x_i - x_j}.$$

Similarly, for x_j we have

$$\prod_{\substack{k=1 \\ k \neq j}}^{N} (x_j - x_k) = (x_j - x_i) \prod_{\substack{k=1 \\ k \neq i,j}}^{N} (x_j - x_k) \quad \Longrightarrow \quad \prod_{\substack{k=1 \\ k \neq i,j}}^{N} (x_j - x_k) = \frac{1}{b_j} \frac{1}{x_j - x_i}.$$

Now, consider the product

$$\prod_{\substack{k=1 \\ k \neq i,j}}^{N} \frac{x_i - x_k}{x_j - x_k} = \frac{\prod\limits_{\substack{k=1 \\ k \neq i,j}}^{N} (x_i - x_k)}{\prod\limits_{\substack{k=1 \\ k \neq i,j}}^{N} (x_j - x_k)} = \frac{\frac{1}{b_i} \frac{1}{x_i - x_j}}{\frac{1}{b_j} \frac{1}{x_j - x_i}} = \frac{b_j}{b_i} \frac{x_j - x_i}{x_i - x_j} = -\frac{b_j}{b_i}.$$

Returning to the expression for $L_j'(x_i)$, we obtain

$$D_{ij} = L_j'(x_i) = \frac{1}{x_j - x_i} \left(-\frac{b_j}{b_i} \right) = -\frac{b_j}{b_i} \frac{1}{x_j - x_i} = \frac{b_j}{b_i} \frac{1}{x_i - x_j}, \quad i \neq j.$$

■

2. We claim that

$$D_{ii} = -\sum_{\substack{j=1 \\ j \neq i}}^{N} D_{ij}. \tag{1.2}$$

Proof:
□ Since the Lagrange basis functions form a partition of unity,

$$\sum_{j=1}^{N} L_j(x) = 1 \quad \text{for all } x,$$

differentiating both sides with respect to x gives

$$\sum_{j=1}^{N} L_j'(x) = 0.$$

In particular, at $x = x_i$ we have

$$\sum_{j=1}^{N} L_j'(x_i) = 0.$$

Writing $D_{ij} = L_j'(x_i)$ yields

$$D_{ii} + \sum_{\substack{j=1\\j\neq i}}^{N} D_{ij} = 0 \Rightarrow D_{ii} = -\sum_{\substack{j=1\\j\neq i}}^{N} D_{ij}.$$

■

■

With everything stated above, the suggested code is the following:

```
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).
        nodes (np.ndarray): The computed LGL nodes on the interval [-1, 1].
        weights (np.ndarray): The computed quadrature weights.
    """

    def __init__(self, p):
        """
        Initialize the LGL object with a given polynomial degree p.

        Parameters:
            p (int): Degree of the Legendre polynomial.
                     (Note: There will be p+1 nodes.)
        """
        self.p = p
        self.nodes, self.weights = self._compute_nodes_weights()

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

            P_p(x) = 1/2^p * sum from k=0 to floor(p/2) of [ (-1)^k * comb(p, k) *
            ↪  comb(2p - 2k, p) * x^(p-2k) ]

```

```
        The coefficients are returned in descending order (highest power first).

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

        Returns:
            np.ndarray: Array of coefficients.
        """
        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.

        The nodes are given by the endpoints -1 and 1 plus the zeros of the
        ↪ derivative
        of the Legendre polynomial P_p(x). The weights are computed via:

            w_i = 2 / (p * (p + 1) * [P_p(x_i)]^2).

        Returns:
            tuple: (nodes, weights) where
                   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: zeros of the derivative P'_p(x)
        interior_nodes = np.sort(dP.r.real)
        # Include endpoints -1 and 1.
        nodes = np.concatenate(([-1.0], interior_nodes, [1.0]))
        # Compute the weights.
```

```python
        weights = 2 / (self.p * (self.p + 1) * (P(nodes) ** 2))
        return nodes, weights

    def get_nodes(self):
        """
        Return the computed LGL nodes.

        Returns:
            np.ndarray: The LGL nodes.
        """
        return self.nodes

    def get_weights(self):
        """
        Return the computed quadrature weights.

        Returns:
            np.ndarray: The quadrature weights.
        """
        return self.weights

    def differentiation_matrix(self):
        """
        Compute the differentiation matrix D whose (i, j)-th entry is given by
            D[i, j] = dL_j/dx (x_i),
        where L_j(x) is the Lagrange basis polynomial corresponding to node x_j.

        Using the barycentric formulation, we first define the barycentric weights:
            b_j = 1 / (product for all k not equal to j of (x_j - x_k)),
        and then for i not equal to j:
            D[i, j] = (b_j / b_i) / (x_i - x_j),
        with the diagonal entries determined by:
            D[i, i] = - (sum for all j not equal to i of D[i, j]).

        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))
        # Compute barycentric weights b_j.
        b = np.zeros(N)
        for j in range(N):
            # np.delete(x, j) removes the j-th element so that the product is taken
            ↪ for all k not equal to j.
            b[j] = 1.0 / np.prod(x[j] - np.delete(x, j))

        # Compute off-diagonal entries.
        for i in range(N):
            for j in range(N):
                if i != j:
                    D[i, j] = (b[j] / b[i]) / (x[i] - x[j])
            # Compute the diagonal entry so that the sum over each row is zero.
```

```python
                D[i, i] = -np.sum(D[i, :])

        return D

def main():
    # Choose p (degree of the Legendre polynomial). There will be p+1 nodes.
    p = 4  # For example, p = 4 (so N = 5 nodes)
    lgl = LGL(p)

    np.set_printoptions(suppress=True)
    print("LGL nodes:")
    print(lgl.get_nodes())

    print("\nLGL quadrature weights:")
    print(lgl.get_weights())

    D = lgl.differentiation_matrix()
    print("\nMatrix D:")
    print(D)

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

by running which we get the following output:

```
LGL nodes:
[-1.         -0.65465367  0.          0.65465367  1.        ]

LGL quadrature weights:
[0.1        0.54444444 0.71111111 0.54444444 0.1       ]

Matrix D:
[[-5.          6.75650249 -2.66666667  1.41016418 -0.5       ]
 [-1.24099025 -0.          1.74574312 -0.76376262  0.25900975]
 [ 0.375      -1.33658458  0.          1.33658458 -0.375     ]
 [-0.25900975  0.76376262 -1.74574312 -0.          1.24099025]
 [ 0.5        -1.41016418  2.66666667 -6.75650249  5.        ]]
```

Problem 2

Code with Python the entries of the differentiation matrix D, mass matrix P and matrix Q.

Solution:

As described in the formulation

$$P = \sum_{\ell=1}^{N} \mathbf{L}(\eta_\ell;\mathbf{x})\,\mathbf{L}(\eta_\ell;\mathbf{x})^T\,\omega_\ell, \quad Q = \sum_{\ell=1}^{N} \mathbf{L}(\eta_\ell;\mathbf{x})\,\frac{d\mathbf{L}}{dx}(\eta_\ell;\mathbf{x})^T\,\omega_\ell,$$

where η_ℓ and ω_ℓ (for $\ell = 1,\ldots,N$) are the LGL (collocation) points and their quadrature weights, and

$$\mathbf{L}(x;\mathbf{x})$$

is the column vector whose components are the Lagrange basis polynomials relative to the discrete nodes $\mathbf{x}$. In our setting the quadrature nodes η_ℓ are the same as the collocation nodes. Recall that for these nodes the Lagrange basis functions satisfy

$$L_j(x_i) = \delta_{ij},$$

that is, when evaluated at the nodes the "Lagrange vector" is the standard basis vector (see Problem 1). Hence, we expect that

$$P = \operatorname{diag}(\omega_1, \omega_2, \ldots, \omega_N)$$

and

$$Q_{ij} = \left(\frac{dL_j}{dx}(x_i)\right)\omega_i,$$

so that

$$D_{ij} = \left[P^{-1}Q\right]_{ij} = \frac{1}{\omega_i}\left(\frac{dL_j}{dx}(x_i)\,\omega_i\right) = \frac{dL_j}{dx}(x_i).$$

With everything stated above, the suggested code is the following:

```python
import numpy as np
import math

class LGL:
    """
    Class for computing Legendre-Gauss-Lobatto (LGL) nodes and quadrature weights.

    Attributes:
        p (int): Degree of the Legendre polynomial (the quadrature has p+1 nodes).
        nodes (np.ndarray): The computed LGL nodes on the interval [-1, 1].
        weights (np.ndarray): The computed quadrature weights.
    """

    def __init__(self, p):
        """
        Initialize the LGL object with a given polynomial degree p.

        Parameters:
            p (int): Degree of the Legendre polynomial.
                     (There will be p+1 nodes.)
```

```python
        """
        self.p = p
        self.nodes, self.weights = self._compute_nodes_weights()

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

            P_p(x) = (1/2^p) * sum from k=0 to floor(p/2) of [ (-1)^k * comb(p,k) * comb(2p-2k,p) * x^(p-2k) ].

        The coefficients are returned in descending order (highest power first).

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

        Returns:
            np.ndarray: Array of coefficients.
        """
        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.

        The nodes are given by the endpoints -1 and 1 plus the zeros of the derivative
        of the Legendre polynomial P_p(x). The weights are computed via:

            w_i = 2 / (p(p+1)[P_p(x_i)]^2).

```

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

    def get_nodes(self):
        """
        Return the computed LGL nodes.

        Returns:
            np.ndarray: The LGL nodes.
        """
        return self.nodes

    def get_weights(self):
        """
        Return the computed quadrature weights.

        Returns:
            np.ndarray: The quadrature weights.
        """
        return self.weights


def compute_matrices_PQD(lgl):
    """
    Using the LGL nodes and weights, compute the matrices P, Q and the differentiation
    matrix D = P^{-1} Q as in equation (13)-(14):

        P = sum_{l=0}^{N-1} L(eta_l;x) L(eta_l;x)^T omega_l,
        Q = sum_{l=0}^{N-1} L(eta_l;x) (dL/dx)(eta_l;x)^T omega_l,

    where:
      - eta_l and omega_l (l = 0,...,N-1) are the LGL nodes and quadrature weights,
      - L(eta_l;x) is the column vector of Lagrange basis polynomials relative to the nodes x,
      - (dL/dx)(eta_l;x) is the column vector of their derivatives.

    In our case the collocation nodes x are the same as the quadrature nodes, so that
    L_j(x_i) = delta_{ij} (the Kronecker delta). Hence, the matrix L is the identity and the
    summation produces:
```

```
      P = diag(omega_0, omega_1, ..., omega_{N-1}),
      Q_{ij} = (dL_j/dx)(x_i) omega_i.

    Then, D = P^{-1} Q recovers the differentiation matrix whose (i,j) entry is
        dL_j/dx(x_i).

    Parameters:
        lgl (LGL): An instance of the LGL class containing nodes and weights.

    Returns:
        P, Q, D (tuple of np.ndarray): The matrices P, Q and the differentiation matrix D.
    """
    nodes = lgl.get_nodes()   # Collocation (and quadrature) nodes
    weights = lgl.get_weights() # Quadrature weights
    N = len(nodes)

    # --- Step 1. Compute barycentric weights for the nodes ---
    # These are used to evaluate the Lagrange basis functions.
    b = np.zeros(N)
    for j in range(N):
        b[j] = 1.0 / np.prod(nodes[j] - np.delete(nodes, j))

    # --- Step 2. Compute the differentiation matrix via the barycentric formula ---
    # That is, for i != j:
    #   D_bary[i,j] = (b[j] / b[i]) / (nodes[i] - nodes[j]),
    # and for the diagonal:
    #   D_bary[i,i] = -sum_{j != i} D_bary[i,j].
    D_bary = np.zeros((N, N))
    for i in range(N):
        for j in range(N):
            if i != j:
                D_bary[i, j] = (b[j] / b[i]) / (nodes[i] - nodes[j])
        D_bary[i, i] = -np.sum(D_bary[i, :])

    # --- Step 3. Form the Lagrange basis matrix L ---
    # For an evaluation at the collocation nodes, the Lagrange basis functions satisfy:
    # L_j(nodes_i) = delta_{ij}. Hence, L is the identity matrix.
    Lmat = np.eye(N)

    # --- Step 4. Compute matrices P and Q via quadrature ---
    P = np.zeros((N, N))
    Q = np.zeros((N, N))
    for l in range(N):
        # L(eta_l;x) is the column vector of Lagrange basis evaluations at eta_l.
        # Since the evaluation is at a collocation node, it is the l-th unit vector.
        L_eta = Lmat[l, :].reshape(N, 1)
        # (dL/dx)(eta_l;x) is taken from the differentiation matrix.
        dL_eta = D_bary[l, :].reshape(N, 1)
        P += L_eta @ L_eta.T * weights[l]
        Q += L_eta @ dL_eta.T * weights[l]

```

```python
    # --- Step 5. Compute D = P^{-1} Q ---
    P_inv = np.linalg.inv(P)
    D = P_inv @ Q
    return P, Q, D

def main():
    # Choose p (degree of the Legendre polynomial). There will be p+1 nodes.
    p = 4  # For example, p = 4 (so N = 5 nodes)
    lgl = LGL(p)

    # Compute matrices P, Q and the differentiation matrix D = P^{-1} Q.
    P, Q, D = compute_matrices_PQD(lgl)

    np.set_printoptions(suppress=True)
    print("LGL nodes:")
    print(lgl.get_nodes())

    print("\nLGL quadrature weights:")
    print(lgl.get_weights())

    print("\nMatrix P:")
    print(P)

    print("\nMatrix Q:")
    print(Q)

    print("\nMatrix D:")
    print(D)

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

by running which we get the following output:

```
LGL nodes:
[-1.         -0.65465367  0.          0.65465367  1.        ]

LGL quadrature weights:
[0.1        0.54444444 0.71111111 0.54444444 0.1       ]

Matrix P:
[[0.1        0.         0.         0.         0.        ]
 [0.         0.54444444 0.         0.         0.        ]
 [0.         0.         0.71111111 0.         0.        ]
 [0.         0.         0.         0.54444444 0.        ]
 [0.         0.         0.         0.         0.1       ]]

Matrix Q:
[[-0.5         0.67565025 -0.26666667  0.14101642 -0.05      ]
 [-0.67565025 -0.          0.95046014 -0.41582631  0.14101642]
 [ 0.26666667 -0.95046014  0.          0.95046014 -0.26666667]
 [-0.14101642  0.41582631 -0.95046014 -0.          0.67565025]
 [ 0.05       -0.14101642  0.26666667 -0.67565025  0.5       ]]
```

```

Matrix D:
[[-5.          6.75650249 -2.66666667  1.41016418 -0.5       ]
 [-1.24099025 -0.          1.74574312 -0.76376262  0.25900975]
 [ 0.375      -1.33658458  0.          1.33658458 -0.375     ]
 [-0.25900975  0.76376262 -1.74574312 -0.          1.24099025]
 [ 0.5        -1.41016418  2.66666667 -6.75650249  5.        ]]
```

GitHub

GitHub link for the codes in the folder `Assignment 1`:
`https://github.com/nurmaton/SBP_KAUST/tree/main/Assignment%201`