Summation by Parts Operators for PDEs

Assignment 3

Abdukhomid Nurmatov

KAUST

06-03-2025

Contents

Problem 1

Explain how to get equation (39) from equation (38) in the paper Entropy-Stable SBP h/p-Nonconforming Discretization for Compressible Flow by Matteo Parsani:

$$\frac{d\mathbf{u}}{dt} + \frac{1}{3}D_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u} + \frac{1}{3}\,\mathrm{diag}(\mathbf{u})D_{x_1}\mathbf{u} = D_{x_1}\boldsymbol{\Theta}, \quad \boldsymbol{\Theta} \equiv D_{x_1}\mathbf{u}. \tag{38}$$

Multiplying (38) by $\mathbf{u}^T P$ results in

$$\frac{1}{2}\frac{d}{dt}\mathbf{u}^T P\mathbf{u} + \frac{1}{3}\left(\mathbf{u}^3(N) - \mathbf{u}^3(1)\right) = \mathbf{u}^T E_{x_1}D_{x_1}\mathbf{u} - \mathbf{u}^T D_{x_1}^T P D_{x_1}\mathbf{u} \tag{39}$$

Solution:

1. **Multiply (38) by $\mathbf{u}^T\mathbf{P}$:**

$$\mathbf{u}^T\mathbf{P}\frac{d\mathbf{u}}{dt} + \frac{1}{3}\mathbf{u}^T\mathbf{P}\left(D_{x_1}\mathrm{diag}(\mathbf{u})\mathbf{u} + \mathrm{diag}(\mathbf{u})D_{x_1}\mathbf{u}\right) = \mathbf{u}^T\mathbf{P}D_{x_1}\boldsymbol{\Theta}.$$

2. **Time Derivative Term:** Since P is symmetric ($P = P^T$), we obtain:

$$\mathbf{u}^T P\frac{d\mathbf{u}}{dt} = \frac{1}{2}\left(\mathbf{u}^T P\frac{d\mathbf{u}}{dt} + \mathbf{u}^T P\frac{d\mathbf{u}}{dt}\right) = \frac{1}{2}\left(\frac{d\mathbf{u}^T}{dt}P^T\mathbf{u} + \mathbf{u}^T P\frac{d\mathbf{u}}{dt}\right) = \frac{1}{2}\frac{d}{dt}(\mathbf{u}^T P\mathbf{u}).$$

3. **Nonlinear Terms:** We now handle the nonlinear terms:

$$\frac{1}{3}\mathbf{u}^T P\left(D_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u} + \mathrm{diag}(\mathbf{u})D_{x_1}\mathbf{u}\right).$$

   Using the SBP property $PD_{x_1} + D_{x_1}^T P^T = E_{x_1} \Rightarrow PD_{x_1} = -D_{x_1}^T P + E_{x_1}$, we rewrite $\mathbf{u}^T PD_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u}$:

$$\mathbf{u}^T PD_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u} = -\mathbf{u}^T D_{x_1}^T P\,\mathrm{diag}(\mathbf{u})\mathbf{u} + \mathbf{u}^T E_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u}.$$

   For $\mathbf{u}^T P\,\mathrm{diag}(\mathbf{u})D_{x_1}\mathbf{u}$ we notice that:

$$\mathbf{u}^T P\,\mathrm{diag}(\mathbf{u})D_{x_1}\mathbf{u} = \mathbf{u}^T D_{x_1}^T P\,\mathrm{diag}(\mathbf{u})\mathbf{u}.$$

   Thus:

$$\frac{1}{3}\mathbf{u}^T P\left(D_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u} + \mathrm{diag}(\mathbf{u})D_{x_1}\mathbf{u}\right) = \frac{1}{3}\mathbf{u}^T E_{x_1}\,\mathrm{diag}(\mathbf{u})\mathbf{u} = \frac{1}{3}\left(\mathbf{u}^3(N) - \mathbf{u}^3(1)\right).$$

4. **Viscous Term:** Finally, we handle the viscous term:

$$\mathbf{u}^T PD_{x_1}\boldsymbol{\Theta}.$$

   Using the SBP property from above:

$$\mathbf{u}^T PD_{x_1}\boldsymbol{\Theta} = -\mathbf{u}^T D_{x_1}^T P\boldsymbol{\Theta} + \mathbf{u}^T E_{x_1}\boldsymbol{\Theta} = -\mathbf{u}^T D_{x_1}^T PD_{x_1}\mathbf{u} + \mathbf{u}^T E_{x_1}D_{x_1}\mathbf{u}.$$

5. **Final Result (Equation 39)** Combining all terms:

$$\frac{1}{2}\frac{d}{dt}(\mathbf{u}^T P\mathbf{u}) + \frac{1}{3}\left(\mathbf{u}^3(N) - \mathbf{u}^3(1)\right) = \mathbf{u}^T E_{x_1}D_{x_1}\mathbf{u} - \mathbf{u}^T D_{x_1}^T PD_{x_1}\mathbf{u}.$$

Problem 2

Write a symbolic code that computes the differentiation matrix D and the flux matrix F using Legendre-Gauss-Lobatto (LGL) nodes.

Solution:

The differentiation matrix D is constructed using the method outlined in Assignment 1. The flux matrix F is defined based on the two-point flux function:

$$F_{ij} = \frac{u_i^2 + u_i u_j + u_j^2}{6}.$$

With everything stated above the suggested code is as follows:

```python
import sympy as sp
import numpy as np

class SymbolicLGL:
    """
    Class to compute Legendre-Gauss-Lobatto (LGL) nodes and differentiation matrix D symbolically.

    Attributes:
        p (int): Degree of the Legendre polynomial (p+1 nodes).
        nodes (list): Computed LGL nodes as symbolic expressions.
        D (sp.Matrix): Differentiation matrix computed symbolically.
    """

    def __init__(self, p):
        """
        Initialize the SymbolicLGL class with polynomial degree p.

        Parameters:
            p (int): Degree of the Legendre polynomial.
        """
        self.p = p
        self.nodes = self.compute_nodes()  # Compute symbolic LGL nodes
        self.D = self.differentiation_matrix()  # Compute symbolic differentiation matrix

    def compute_nodes(self):
        """
        Compute the LGL nodes as symbolic expressions.

        Returns:
            list: LGL nodes including -1 and 1 with symbolic interior nodes.
        """
        x = sp.Symbol('x')  # Define symbolic variable
        P = sp.legendre(self.p, x)  # Legendre polynomial P_p(x)
        dP = sp.diff(P, x)  # Compute derivative P'_p(x)

        # Solve for roots of P'_p(x) to get interior nodes
```

```python
        nodes = sorted(sp.solveset(dP, x, domain=sp.Interval(-1, 1)))

        # LGL nodes include -1 and 1
        return [-1] + nodes + [1]

    def differentiation_matrix(self):
        """
        Compute the symbolic differentiation matrix D using the barycentric formula.

        Returns:
            sp.Matrix: Symbolic differentiation matrix.
        """
        x = self.nodes  # LGL nodes
        N = len(x)  # Number of nodes
        D = sp.zeros(N, N)  # Initialize symbolic differentiation matrix

        # Compute barycentric weights: b_j = 1 / (product for all k not equal to j of (x_j - x_k)).
        b = [1 / np.prod([x[j] - x[k] for k in range(N) if k != j]) for j in range(N)]

        # Compute differentiation matrix 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])
            D[i, i] = -sum(D[i, :])  # Ensure row sum is zero

        return D

def compute_flux_matrix(nodes):
    """
    Compute the symbolic flux matrix F based on the given LGL nodes.

    The flux function is defined as:
        F_ij = (u_i^2 + u_i u_j + u_j^2) / 6

    Parameters:
        nodes (list): LGL nodes.

    Returns:
        tuple: (F, u) where F is the symbolic flux matrix, and u is the list of symbolic variables.
    """
    N = len(nodes)  # Number of nodes
    u = sp.symbols(f'u1:{N+1}')  # Define symbolic variables u1, u2, ..., uN
    F = sp.zeros(N, N)  # Initialize symbolic flux matrix

    # Compute flux values using the given formula
    for i in range(N):
        for j in range(N):
            F[i, j] = (u[i]**2 + u[i]*u[j] + u[j]**2) / 6
```

```python
    return F, u

def main():
    """
    Main function to compute and display symbolic and numerical matrices.
    """
    p = 2  # Example polynomial degree
    lgl = SymbolicLGL(p)  # Create LGL object

    # Compute differentiation matrix D
    D_symbolic = lgl.D  # Symbolic matrix
    D_numeric = np.array(D_symbolic.evalf().tolist(), dtype=np.float64)  # Numeric
    ↪  conversion

    # Compute flux matrix F
    F_symbolic, u = compute_flux_matrix(lgl.nodes)

    # Print results
    print("\nSymbolic Differentiation Matrix D:")
    sp.pprint(D_symbolic)

    print("\nNumerical Differentiation Matrix D:")
    np.set_printoptions(precision=6, suppress=True)  # Format output
    print(D_numeric)

    print("\nSymbolic Flux Matrix F:")
    sp.pprint(F_symbolic)

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

by running which we get the following output:

```
Symbolic Differentiation Matrix D:
⎡-3/2  2   -1/2⎤
⎢              ⎥
⎢-1/2  0   1/2 ⎥
⎢              ⎥
⎣1/2   -2  3/2 ⎦

Numerical Differentiation Matrix D:
[[-1.5  2.  -0.5]
 [-0.5  0.   0.5]
 [ 0.5 -2.   1.5]]

Symbolic Flux Matrix F:
⎡         2            2            2    2             2⎤
⎢       u₁           u₁     u₁⋅u₂  u₂   u₁     u₁⋅u₃  u₃⎥
⎢       ───          ─── + ───── + ───  ─── + ───── + ───⎥
⎢        2            6      6      6    6      6     6 ⎥
⎢                                                        ⎥
⎢  2                 2           2           2             2⎥
⎢u₁     u₁⋅u₂  u₂          u₂          u₂     u₂⋅u₃  u₃⎥
⎢─── + ───── + ───         ───         ─── + ───── + ───⎥
⎢ 6      6      6           2           6      6      6 ⎥
⎢                                                        ⎥
⎢  2             2    2             2           2       ⎥
⎢u₁     u₁⋅u₃  u₃   u₂     u₂⋅u₃  u₃          u₃       ⎥
⎢─── + ───── + ───  ─── + ───── + ───         ───       ⎥
⎣ 6      6      6    6      6      6           2        ⎦
○ (base) abdulhamid@macbookpro Downloads %
```

GitHub

GitHub link for the codes in the folder `Assignment 3`:
`https://github.com/nurmaton/SBP_KAUST/tree/main/Assignment%203`