Summation by Parts Operators for PDEs

Assignment 5

Abdukhomid Nurmatov

KAUST

12-04-2025

Contents

Problem 1

Implement a Python subroutine using the `sympy` library to generate symbolic expressions for the two-point entropy conservative flux vector $\tilde{\mathbf{f}}_{S,j}$ and its constituent averaged parameters $(\hat{u}_k, \hat{p}, \hat{\rho}, \hat{h}, \hat{H}, \theta_1, \theta_2)$. The implementation should be based on the mathematical formulas provided in Section 4.2.1, "Affordable entropy consistent Euler flux", of the work by Parsani, Carpenter, and Nielsen (*Entropy stable wall boundary conditions for the three-dimensional compressible Navier–Stokes equations*, Journal of Computational Physics 292 (2015) 88–113).

Solution:

So, we need to describe the symbolic formulation of the two-point entropy conservative flux, denoted by $\tilde{\mathbf{f}}_{S,j}$, for the inviscid terms of the compressible Navier-Stokes equations. The specific formulas presented here are detailed in Section 4.2.1, "Affordable entropy consistent Euler flux", of the work by Parsani, Carpenter, and Nielsen (*Entropy stable wall boundary conditions for the three-dimensional compressible Navier–Stokes equations*, Journal of Computational Physics 292 (2015) 88–113). The goal is to represent these potentially complex expressions symbolically using computational tools like Python's `sympy` library.

The flux vector $\tilde{\mathbf{f}}_{S,j}$ in the j-th spatial direction ($j \in \{1, 2, 3\}$) between states at indices i and $i+1$ is given by:

$$\tilde{\mathbf{f}}_{S,j}(q_i, q_{i+1}) = \begin{pmatrix} \hat{\rho}\hat{u}_j \\ \hat{\rho}\hat{u}_j\hat{u}_1 + \delta_{j1}\hat{p} \\ \hat{\rho}\hat{u}_j\hat{u}_2 + \delta_{j2}\hat{p} \\ \hat{\rho}\hat{u}_j\hat{u}_3 + \delta_{j3}\hat{p} \\ \hat{\rho}\hat{u}_j\hat{H} \end{pmatrix} \tag{1}$$

where $q = (\rho, \rho u_1, \rho u_2, \rho u_3, \rho E)^T$ represents the vector of conserved variables, δ_{jk} is the Kronecker delta, and the hat quantities $(\hat{\cdot})$ represent specific averages between states i and $i+1$.

The averaged quantities are defined as follows:

- **Averaged Velocity ($\hat{\mathbf{u}} = (\hat{u}_1, \hat{u}_2, \hat{u}_3)$) and Pressure ($\hat{p}$):** These use a weighted average based on the inverse square root of temperature (T):

$$\hat{u}_k = \frac{u_{k,i}/\sqrt{T_i} + u_{k,i+1}/\sqrt{T_{i+1}}}{1/\sqrt{T_i} + 1/\sqrt{T_{i+1}}}, \quad k \in \{1, 2, 3\} \tag{2}$$

$$\hat{p} = \frac{p_i/\sqrt{T_i} + p_{i+1}/\sqrt{T_{i+1}}}{1/\sqrt{T_i} + 1/\sqrt{T_{i+1}}} \tag{3}$$

- **Density-related Average ($\hat{\rho}$):** This average involves a logarithmic mean structure:

$$\hat{\rho} = \frac{\left(\frac{1}{\sqrt{T_i}} + \frac{1}{\sqrt{T_{i+1}}}\right)(\sqrt{T_i}\rho_i - \sqrt{T_{i+1}}\rho_{i+1})}{2(\log(\sqrt{T_i}\rho_i) - \log(\sqrt{T_{i+1}}\rho_{i+1}))} \tag{4}$$

  In the limit $\sqrt{T_i}\rho_i \to \sqrt{T_{i+1}}\rho_{i+1}$, the value becomes $\hat{\rho} \to \frac{1}{2}\left(\frac{1}{\sqrt{T_i}} + \frac{1}{\sqrt{T_{i+1}}}\right)\sqrt{T_i}\rho_i$.

- **Auxiliary Parameters (θ_1, θ_2):**

$$\theta_1 = \frac{\sqrt{T_i}\rho_i + \sqrt{T_{i+1}}\rho_{i+1}}{\left(\frac{1}{\sqrt{T_i}} + \frac{1}{\sqrt{T_{i+1}}}\right)(\sqrt{T_i}\rho_i - \sqrt{T_{i+1}}\rho_{i+1})} \tag{5}$$

$$\theta_2 = \frac{\frac{\gamma+1}{\gamma-1}\log\left(\frac{\sqrt{T_{i+1}}}{\sqrt{T_i}}\right)}{\log\left(\frac{\sqrt{T_i}\rho_i}{\sqrt{T_{i+1}}\rho_{i+1}}\right)\left(\frac{1}{\sqrt{T_i}} - \frac{1}{\sqrt{T_{i+1}}}\right)} \tag{6}$$

where R is the specific gas constant and γ is the ratio of specific heats. Note potential singularities in the denominators.

- **Averaged Specific Enthalpy ($\hat{h}$):**

$$\hat{h} = \frac{R\log\left(\frac{\sqrt{T_i}\rho_i}{\sqrt{T_{i+1}}\rho_{i+1}}\right)(\theta_1 + \theta_2)}{\left(\frac{1}{\sqrt{T_i}} + \frac{1}{\sqrt{T_{i+1}}}\right)} \tag{7}$$

Note potential singularities related to θ_1, θ_2 and the logarithm term.

- **Averaged Total Enthalpy ($\hat{H}$):** This is derived from the specific enthalpy and averaged velocity:

$$\hat{H} = \hat{h} + \frac{1}{2}(\hat{u}_1^2 + \hat{u}_2^2 + \hat{u}_3^2) \tag{8}$$

The Python function `get_symbolic_entropy_flux` implements these symbolic definitions using the `sympy` library, returning the flux vector $\tilde{\mathbf{f}}_{S,j}$ and a dictionary containing the intermediate parameters. With everything stated above the suggested code is as follows:

```
import sympy

def get_symbolic_entropy_flux(j):
    """
    Generates symbolic expressions for the two-point entropy conservative flux
    vector f~_{S,j} and related intermediate parameters based on the formulas
    provided (cf. Ismail and Roe, 2009; Parsani et al., 2015).

    Args:
        j (int): The spatial direction index (1, 2, or 3).

    Returns:
        tuple: A tuple containing:
            - F_vector (sympy.Matrix): The symbolic flux vector f~_{S,j}.
            - params (dict): A dictionary containing symbolic expressions for
                             intermediate parameters (û, p^, ĥ, Ĥ, ρ^, θ₁, θ₂).
                             'u_hat' is a list of 3 components.

    Raises:
        ValueError: If j is not 1, 2, or 3.

    Notes:
        - The formulas for θ₁, θ₂, and ĥ involve terms that can lead to
```

```
            division by zero or indeterminate forms (0/0) in specific limits
            (e.g., T_i = T_{i+1}, √(T_i ρ_i) = √(T_{i+1} ρ_{i+1})). The direct symbolic
            implementation here may require careful numerical handling or
            limit analysis in a practical application.
    """
    if j not in [1, 2, 3]:
        raise ValueError("Spatial direction index j must be 1, 2, or 3")

    # --- Define Base Symbolic Variables ---
    rho_i, rho_ip1 = sympy.symbols('rho_i rho_{i+1}', positive=True)
    T_i, T_ip1 = sympy.symbols('T_i T_{i+1}', positive=True) # Temperature
    p_i, p_ip1 = sympy.symbols('p_i p_{i+1}')             # Pressure

    # Velocity components at i and i+1
    u1_i, u2_i, u3_i = sympy.symbols('u1_i u2_i u3_i')
    u1_ip1, u2_ip1, u3_ip1 = sympy.symbols('u1_{i+1} u2_{i+1} u3_{i+1}')

    u_vec_i = [u1_i, u2_i, u3_i]
    u_vec_ip1 = [u1_ip1, u2_ip1, u3_ip1]

    # Thermodynamic constants
    R, gamma = sympy.symbols('R gamma', positive=True) # Gas constant, ratio of specific heats

    # --- Calculate Intermediate Parameters ---

    # Square roots of Temperature and their inverses
    sqrt_T_i = sympy.sqrt(T_i)
    sqrt_T_ip1 = sympy.sqrt(T_ip1)
    sqrt_T_i_inv = 1 / sqrt_T_i
    sqrt_T_ip1_inv = 1 / sqrt_T_ip1

    # Common denominator term for averages
    avg_denom = sqrt_T_i_inv + sqrt_T_ip1_inv

    # Parameter û (Averaged velocity vector) [u1_hat, u2_hat, u3_hat]
    u_hat_vec = []
    for k in range(3):
        u_hat_k = (u_vec_i[k] * sqrt_T_i_inv + u_vec_ip1[k] * sqrt_T_ip1_inv) / avg_denom
        u_hat_vec.append(u_hat_k)
    u1_hat, u2_hat, u3_hat = u_hat_vec
    u_hat_sq_norm = sum(comp**2 for comp in u_hat_vec) # Needed for H_hat

    # Parameter p^ (Averaged pressure)
    p_hat = (p_i * sqrt_T_i_inv + p_ip1 * sqrt_T_ip1_inv) / avg_denom

    # Parameters involving sqrt(T)*rho
    sqrt_T_rho_i = sqrt_T_i * rho_i
    sqrt_T_rho_ip1 = sqrt_T_ip1 * rho_ip1

    # Define log terms carefully
    log_sqrt_T_rho_i = sympy.log(sqrt_T_rho_i)
```

```
    log_sqrt_T_rho_ip1 = sympy.log(sqrt_T_rho_ip1)
    log_sqrt_T_ip1_over_T_i = sympy.log(sqrt_T_ip1 / sqrt_T_i) # = 0.5 * log(T_{i+1}/T_i)

    # Parameter θ1
    theta1_num = sqrt_T_rho_i + sqrt_T_rho_ip1
    theta1_den_term2 = (sqrt_T_rho_i - sqrt_T_rho_ip1)
    theta1_den = avg_denom * theta1_den_term2
    theta1 = theta1_num / theta1_den # Note potential limit issues


    # Parameter θ2
    theta2_num = (gamma + 1) / (gamma - 1) * log_sqrt_T_ip1_over_T_i
    theta2_den_log_term = log_sqrt_T_rho_i - log_sqrt_T_rho_ip1
    theta2_den_diff_term = sqrt_T_i_inv - sqrt_T_ip1_inv
    theta2_den = theta2_den_log_term * theta2_den_diff_term
    theta2 = theta2_num / theta2_den # Note potential limit issues


    # Parameter ĥ (Averaged specific enthalpy) - Using the explicit formula
    h_hat_log_term = log_sqrt_T_rho_i - log_sqrt_T_rho_ip1
    h_hat_num = R * h_hat_log_term * (theta1 + theta2)
    h_hat_den = avg_denom
    h_hat = h_hat_num / h_hat_den # Note potential limit issues


    # Parameter Ĥ (Averaged total enthalpy) - Derived from h_hat
    H_hat = h_hat + 0.5 * u_hat_sq_norm


    # Parameter ρ^ (Logarithmic mean related density)
    rho_hat_num = avg_denom * (sqrt_T_rho_i - sqrt_T_rho_ip1)
    rho_hat_den = 2 * (log_sqrt_T_rho_i - log_sqrt_T_rho_ip1)
    # Define using Piecewise for the limit of LogMean(X,Y) as Y->X is X.
    rho_hat_limit = avg_denom * sqrt_T_rho_i / 2
    rho_hat = sympy.Piecewise(
        (rho_hat_limit, sympy.Eq(rho_hat_den, 0)), # Handles sqrt_T_rho_i = sqrt_T_rho_ip1
        (rho_hat_num / rho_hat_den, True)
    )

    # --- Assemble the Flux Vector f~_{S,j} ---
    u_hat_j = u_hat_vec[j-1] # Select the j-th component of û (0-indexed list)

    # Kronecker deltas
    delta_j1 = 1 if j == 1 else 0
    delta_j2 = 1 if j == 2 else 0
    delta_j3 = 1 if j == 3 else 0

    # Flux components
    f_tilde_1 = rho_hat * u_hat_j
    f_tilde_2 = rho_hat * u_hat_j * u1_hat + delta_j1 * p_hat
    f_tilde_3 = rho_hat * u_hat_j * u2_hat + delta_j2 * p_hat
```

```python
    f_tilde_4 = rho_hat * u_hat_j * u3_hat + delta_j3 * p_hat
    f_tilde_5 = rho_hat * u_hat_j * H_hat

    # Create the symbolic matrix (vector)
    F_vector = sympy.Matrix([f_tilde_1, f_tilde_2, f_tilde_3, f_tilde_4, f_tilde_5])

    # Store intermediate parameters in a dictionary for potential reuse
    params = {
        'u_hat': u_hat_vec,    # List [u1_hat, u2_hat, u3_hat]
        'p_hat': p_hat,
        'rho_hat': rho_hat,
        'theta1': theta1,
        'theta2': theta2,
        'h_hat': h_hat,        # Avg. specific enthalpy (calculated from formula)
        'H_hat': H_hat,        # Avg. total enthalpy (derived from h_hat)
    }

    return F_vector, params

# --- Example Usage ---
if __name__ == "__main__":
    # Initialize sympy for nice printing in console
    sympy.init_printing(use_unicode=True)

    print("Calculating symbolic flux for j=1:")
    try:
        F1_vector, parameters1 = get_symbolic_entropy_flux(j=1)

        print("\nFlux Vector f~_{S,1}:")
        sympy.pprint(F1_vector)

        print("\n--- Intermediate Parameters ---")
        for name, expr in parameters1.items():
            print(f"\nParameter: {name}")
            if isinstance(expr, list): # Print vector components
                for i, comp in enumerate(expr):
                    print(f"  Component {i+1}:")
                    sympy.pprint(comp)
            else:
                sympy.pprint(expr)

    except ValueError as e:
        print(f"Error: {e}")

    # # Example for numerical substitution (requires defining values)
    # R_val, gamma_val = 287.0, 1.4 # Example values for air
    # values = {
    #     'rho_i': 1.2, 'rho_{i+1}': 1.0,
    #     'T_i': 300, 'T_{i+1}': 280,
    #     'p_i': 101325, 'p_{i+1}': 95000,
    #     'u1_i': 50, 'u2_i': 10, 'u3_i': 0,
    #     'u1_{i+1}': 40, 'u2_{i+1}': 5, 'u3_{i+1}': 0,
    #     'R': R_val, 'gamma': gamma_val
```

```
    # }
    # F1_numerical = F1_vector.subs(values).evalf()
    # print("\nExample Numerical Flux Vector (j=1):")
    # sympy.pprint(F1_numerical)
```

GitHub

GitHub link for the codes in the folder `Assignment 5`:
`https://github.com/nurmaton/SBP_KAUST/tree/main/Assignment%205`