Summation by Parts Operators for PDEs

Assignment 4

Abdukhomid Nurmatov

KAUST

23-03-2025

Contents

Problem 1

Prepare the code infrastructure for solving a time-dependent problem in N_{dim} dimensions and N_{eqs} equations. Ensure the following components are implemented:

1. Setup of grid parameters:
   - Define domain and number of cells.
2. Setup of SBP matrices for spatial discretization of a PDE with first and second derivatives:
   - Specify solution polynomial order.
   - Compute LGL points, D, P, Q, etc.
3. Implement a generic function to set the initial condition.
4. Implement a generic function to set periodic boundary conditions.
5. Implement a subroutine/function that computes the derivative at LGL points given function values at those points.
6. Implement a function/subroutine to solve a system of ODEs of generic size using the following methods:
   - Forward Euler
   - Heun method
   - Classical 4-stage 4th-order Runge-Kutta scheme
   - Backward Euler
7. Ensure all necessary for-loops are included.

Solution:

This problem builds upon the previous assignments and provides a solid and extensible infrastructure for solving time-dependent PDEs using high-order SBP methods.

SBP Matrices Using LGL Nodes

This part of the code (inside the `LGL_SBP` class) sets up the SBP (Summation–by–Parts) discretization on the reference interval $[-1, 1]$. It computes:

- **LGL Nodes and Quadrature Weights:**
  The code computes the Legendre–Gauss–Lobatto (LGL) nodes, which are the endpoints -1 and 1 along with the interior nodes obtained as the zeros of the derivative of the Legendre polynomial. These nodes provide high accuracy for polynomial interpolation and quadrature.
- **Differentiation Matrix D:**
  Using a barycentric formulation, the code computes the differentiation matrix D whose (i, j)-th entry is given by
  $$D_{ij} = \begin{cases} \dfrac{b_j}{b_i} \dfrac{1}{x_i - x_j}, & i \neq j, \\ -\sum_{j \neq i} D_{ij}, & i = j, \end{cases}$$
  where the barycentric weights b_j are computed from the product of differences between nodes. This is the same approach used in Assignments 1 and 2.
- **Mass Matrix P and Matrix Q:**
  In addition, the mass matrix P (a diagonal matrix of quadrature weights) and the matrix Q

(where each entry is $Q_{ij} = D_{ij}\, w_i$) are computed. These matrices help enforce the SBP property $D = P^{-1}Q$ and are used in the theoretical analysis of stability. (See Assignment 1 Problem 1 and Assignment 2 Problem 1 for more details on how these matrices are derived.)

Grid and SBP Assembly in 1D

The `Grid1D` class builds the physical grid by dividing the computational domain $[a, b]$ into a specified number of cells. For each cell:

- **Cell Boundaries:**
  The domain is split uniformly, and cell boundaries are stored.
- **Mapping from Reference to Physical Element:**
  Each cell's collocation points are obtained by applying an affine mapping from the reference element $[-1, 1]$ to the physical cell $[x_{\text{left}}, x_{\text{right}}]$. The mapping is given by

  $$y = \frac{x_{\text{right}} - x_{\text{left}}}{2}(x + 1) + x_{\text{left}},$$

  where x are the LGL nodes on $[-1, 1]$. This approach is similar to what was described in Assignment 2 Problem 1.
- **Assembly of Global Nodes:**
  The collocation points from all cells are stored, and note that the cell interfaces appear twice—once as the right endpoint of one cell and once as the left endpoint of the next.

Initial Condition

The `set_initial_condition` method in `Grid1D` accepts a function (for example, $u(x) = \sin(2\pi x)$) and applies it to every cell's physical nodes. This results in a multidimensional array with shape (num_cells, $p + 1$) representing the solution at time $t = 0$. This method is generic so that it can be used for any given initial condition.

Periodic Boundary Conditions

The `apply_periodic_bc` method enforces periodic boundary conditions. Because the grid is built cell-by-cell, the interfaces between cells are represented twice. To enforce continuity and periodicity:

- **Interface Averaging:**
  The code averages the duplicate degrees of freedom at the interface between adjacent cells.
- **Domain Boundaries:**
  It then enforces periodicity at the boundaries by averaging the first node of the first cell with the last node of the last cell. This approach is a simplified way of ensuring that the solution is continuous across the domain—a concept that is critical in many high-order discretization schemes.

Spatial Derivative Computation

The function `compute_spatial_derivative` calculates the spatial derivative u_x in each cell by applying the differentiation matrix. For each cell:

- **Scaling the Differentiation Matrix:**
  Because the differentiation matrix D was computed on the reference element $[-1, 1]$, it must be scaled by the factor $\frac{2}{\Delta x}$ (where Δx is the cell width) to obtain the derivative in the physical domain, see Assignment 2 Problem 1.

- **Application:**
  The derivative in each cell is then computed by taking the dot product of the scaled matrix with the solution vector in that cell.

ODE Solvers for Time Integration

The `solve_ode` function integrates the semi-discrete system of ODEs in time. This function is generic and supports several time-stepping methods:

- Forward Euler (FE)
- Heun's Method (a predictor-corrector approach)
- Classical RK4 (fourth-order Runge-Kutta)
- Backward Euler (BE) using fixed-point iteration

The function flattens the spatial grid solution, advances it in time using the chosen method, and then reshapes it back to the grid structure.

Overall Structure and Animation

Finally, in the `main()` function:

1. **Grid and Problem Setup:**
   The domain is defined, and a grid is created by dividing it into cells. SBP matrices are computed on the reference element and then mapped to each cell.
2. **Initial Condition & Periodic BC:**
   The initial condition is set (e.g., $u(x) = \sin(2\pi x)$), and periodic boundary conditions are applied.
3. **ODE System Definition:**
   For demonstration, the linear advection equation $u_t = -u_x$ is used. The spatial derivative is computed via the previously defined function.
4. **Time Integration:**
   The system is evolved in time using one of the provided methods (in the code, the Backward Euler method is chosen).
5. **Animation:**
   The evolving solution is animated using Matplotlib's `FuncAnimation`. Each cell's solution is plotted as a separate line, and the plot updates to reflect the current time level.

With everything stated above the suggested code is as follows:

```
import numpy as np
import math
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

##############################################
# 1. SBP Matrices using LGL nodes (Reference)
##############################################
class LGL_SBP:
    """
    Computes the Legendre-Gauss-Lobatto nodes, quadrature weights, and the
```

```
        differentiation matrix D (and derived matrices P, Q) on the reference element
        ↪  [-1,1].
        """
        def __init__(self, p):
            self.p = p
            self.nodes, self.weights = self._compute_nodes_weights()
            self.D = self._differentiation_matrix()
            self.P = np.diag(self.weights)
            # Compute Q: Q[i,j] = D[i,j] * weights[i]
            self.Q = np.zeros((p+1, p+1))
            for i in range(p+1):
                for j in range(p+1):
                    self.Q[i, j] = self.D[i, j] * self.weights[i]

        @staticmethod
        def legendre_poly_coeffs(p):
            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):
            coeffs = cls.legendre_poly_coeffs(p)
            return np.poly1d(coeffs)

        def _compute_nodes_weights(self):
            P_poly = self.legendre_poly(self.p)
            dP = P_poly.deriv()
            # The interior nodes are the zeros of P'_p(x)
            interior_nodes = np.sort(dP.r.real)
            # Include endpoints -1 and 1.
            nodes = np.concatenate(([-1.0], interior_nodes, [1.0]))
            # Quadrature weights on [-1,1]:
            weights = 2 / (self.p * (self.p + 1) * (P_poly(nodes)**2))
            return nodes, weights

        def _differentiation_matrix(self):
            x = self.nodes
            N = len(x)
            D = np.zeros((N, N))
            b = np.zeros(N)
            # Compute barycentric weights:
            for j in range(N):
                b[j] = 1.0 / np.prod(x[j] - np.delete(x, j))
            # 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])
```

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

##############################################
# 2. Grid and SBP assembly in 1D (Cell-Based)
##############################################
class Grid1D:
    """
    Sets up the spatial grid for a 1D problem.
    Divides the domain [a, b] into a given number of cells.
    Each cell is discretized using LGL collocation points (SBP).
    """
    def __init__(self, a, b, num_cells, p, Neqs=1):
        self.a = a
        self.b = b
        self.num_cells = num_cells
        self.p = p           # Polynomial order (each cell has p+1 points)
        self.Neqs = Neqs     # Number of equations (system size per collocation point)

        # Compute cell boundaries (uniform grid)
        self.cell_boundaries = np.linspace(a, b, num_cells + 1)
        # Setup SBP matrices on the reference cell [-1,1]
        self.sbp = LGL_SBP(p)
        # Assemble global grid: for each cell, map the reference LGL nodes to the physical cell.
        self.global_nodes = []  # list of arrays (each of shape (p+1,))
        self.cell_sizes = []
        for i in range(num_cells):
            x_left = self.cell_boundaries[i]
            x_right = self.cell_boundaries[i+1]
            self.cell_sizes.append(x_right - x_left)
            # Mapping: y = ((x_right - x_left)/2) * (xi + 1) + x_left
            y = ((x_right - x_left)/2.0) * (self.sbp.nodes + 1) + x_left
            self.global_nodes.append(y)
        # Optionally, assemble a single vector of all nodes (note: cell interfaces appear twice)
        self.all_nodes = np.concatenate(self.global_nodes)

    def set_initial_condition(self, init_func):
        """
        Set the initial condition on the grid.
        init_func: callable that accepts an array of coordinates and returns the initial values.
        Returns an array of shape (num_cells, p+1).
        """
        u0 = []
        for cell in self.global_nodes:
            u0.append(init_func(cell))
        u0 = np.array(u0)
        return u0

    def apply_periodic_bc(self, u):
```

```
        """
        Enforce periodic boundary conditions.

        Here, since the grid is assembled cell-by-cell,
        the interfaces between cells appear twice (once as the right endpoint of one cell
        and once as the left endpoint of the next cell). To enforce periodicity,
        we average the duplicate values at each interface and ensure the first and last
        nodes are consistent.
        """
        u_new = u.copy()
        num_cells = self.num_cells

        # For internal interfaces: average the right boundary of cell i with the left boundary of cell i+1.
        for i in range(num_cells - 1):
            avg = 0.5 * (u_new[i, -1] + u_new[i+1, 0])
            u_new[i, -1] = avg
            u_new[i+1, 0] = avg

        # For the periodic boundary at the domain boundaries:
        avg = 0.5 * (u_new[-1, -1] + u_new[0, 0])
        u_new[-1, -1] = avg
        u_new[0, 0] = avg

        return u_new

##############################################
# 3. Compute Spatial Derivative at LGL points
##############################################
def compute_spatial_derivative(u, grid):
    """
    Compute the spatial derivative in each cell.
    u: array of shape (num_cells, p+1) containing solution values at collocation points.
    Returns an array of the same shape containing du/dx.
    """
    num_cells = grid.num_cells
    p = grid.p
    du = np.zeros_like(u)
    for i in range(num_cells):
        # For cell i, map from [-1,1] to physical cell of length dx
        dx = grid.cell_sizes[i]
        # Scale the reference differentiation matrix:
        D_cell = (2 / dx) * grid.sbp.D
        # Compute derivative in cell i
        for j in range(p+1):
            du[i, j] = np.dot(D_cell[j, :], u[i, :])
    return du

##############################################
# 4. ODE Solvers for Time Integration
```

```python
##############################################
def solve_ode(f, u0, t0, tf, dt, method="RK4", max_iter_BE=50, tol_BE=1e-6):
    """
    Solve du/dt = f(t,u) with initial condition u0 from t0 to tf using step dt.
    Supported methods: "FE" (Forward Euler), "Heun", "RK4", "BE" (Backward Euler).
    Returns time levels and a list of solution arrays (with same shape as u0).
    """
    u_shape = u0.shape
    u = u0.flatten()
    t_values = [t0]
    u_values = [u.copy()]
    t = t0
    while t < tf - 1e-12:
        if t + dt > tf:
            dt = tf - t
        if method == "FE":
            u = u + dt * f(t, u)
        elif method == "Heun":
            f_n = f(t, u)
            u_predict = u + dt * f_n
            f_np1 = f(t+dt, u_predict)
            u = u + dt/2.0 * (f_n + f_np1)
        elif method == "RK4":
            k1 = f(t, u)
            k2 = f(t + dt/2.0, u + dt/2.0 * k1)
            k3 = f(t + dt/2.0, u + dt/2.0 * k2)
            k4 = f(t + dt, u + dt * k3)
            u = u + dt/6.0 * (k1 + 2*k2 + 2*k3 + k4)
        elif method == "BE":
            u_new = u.copy()
            for _ in range(max_iter_BE):
                u_prev = u_new.copy()
                u_new = u + dt * f(t+dt, u_new)
                if np.linalg.norm(u_new - u_prev) < tol_BE:
                    break
            u = u_new
        else:
            raise ValueError("Unknown time integration method.")
        t += dt
        t_values.append(t)
        u_values.append(u.copy())
    u_values = [u_vec.reshape(u_shape) for u_vec in u_values]
    return np.array(t_values), u_values

##############################################
# 5. Main: Assemble, Solve, and Animate a 1D Problem
##############################################
def main():
    # Grid and problem parameters:
    a = 0.0
    b = 1.0
    num_cells = 10       # number of cells
    p = 4                # polynomial order (each cell has 5 points)
```

```
    Neqs = 1             # scalar problem

    # Create grid:
    grid = Grid1D(a, b, num_cells, p, Neqs)

    # Print SBP matrices (from reference element):
    print("Mass matrix P on reference element [-1,1]:")
    print(grid.sbp.P)
    print("\nMatrix Q on reference element [-1,1]:")
    print(grid.sbp.Q)

    # Set initial condition (e.g., u(x) = sin(2*pi*x)):
    def init_func(x):
        return np.sin(2*np.pi*x)
    u0 = grid.set_initial_condition(init_func)

    # Apply periodic boundary conditions:
    u0 = grid.apply_periodic_bc(u0)

    # Define the ODE system (linear advection: u_t = -u_x)
    # (For demonstration, we use a constant speed and simple derivative computation.)
    def ode_system(t, u_vec):
        u_mat = u_vec.reshape(u0.shape)
        du = compute_spatial_derivative(u_mat, grid)
        return (-du).flatten()

    # For testing purposes, you might uncomment one of the following:
    # def ode_system(t, u_vec):
    #     # u_t = 0 everywhere (stationary solution)
    #     return np.zeros_like(u_vec)
    # def ode_system(t, u_vec):
    #     # u_t = 1 everywhere (solution should grow linearly in time)
    #     return np.ones_like(u_vec)

    # Time integration parameters:
    t0 = 0.0
    tf = 0.5
    dt = 0.001
    # Solve using RK4 (alternatively "FE", "Heun", or "BE")
    t_values, u_values = solve_ode(ode_system, u0, t0, tf, dt, method="BE")

    # Animation: Plot each cell's solution as a separate line.
    fig, ax = plt.subplots()
    ax.set_xlim(a, b)
    ax.set_ylim(-1.5, 1.5)
    ax.set_xlabel("x")
    ax.set_ylabel("u")
    ax.set_title("Time-dependent solution (linear advection)")

    # Create a list of Line2D objects, one per cell.
    lines = []
    for cell in grid.global_nodes:
        (line,) = ax.plot(cell, np.zeros_like(cell), 'b.-')
```

```
        lines.append(line)

    # Update function for the animation:
    def update(frame):
        sol = u_values[frame]  # shape: (num_cells, p+1)
        for i, line in enumerate(lines):
            line.set_data(grid.global_nodes[i], sol[i, :])
        ax.set_title(f"Time t = {t_values[frame]:.3f}")
        return lines

    anim = FuncAnimation(fig, update, frames=len(t_values), interval=50, blit=False)
    plt.show()

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

Instability Without Proper Interface Treatment:
In a multi-element SBP method without specialized interface treatment, each cell solves the PDE locally and only averages (or copies) the interface values. While this forces the solution u to be continuous at the interfaces, it does *not* ensure that the flux (i.e., the derivative u_x) is consistent with the PDE $u_t + u_x = 0$. As a result:

1. **Blow-Up / Instability:** Incorrect interface coupling may cause each cell's local solution to push the interface values in a way that diverges from the true wave propagation, leading to growing oscillations or blow-up.

2. **Tearing at Interfaces:** Even if blow-up is avoided, the solution can exhibit visible discontinuities (tears) at cell boundaries because matching u values alone does not guarantee the correct flux.

Why Averaging Is Insufficient:

- Averaging does not incorporate the PDE dynamics (e.g., wave speed and flux continuity).
- Each cell evolves independently using its local differentiation matrix; without proper coupling, the neighboring cells do not exchange the necessary flux information.

SAT (Simultaneous Approximation Term) Approach to Fix the Issue:

- **Local Derivative:** Each cell computes its interior derivative using the SBP differentiation matrix.
- **Penalty Terms:** At cell interfaces, SAT terms penalize the jump in u (e.g., add a term proportional to $(u_i - u_{i+1})$) to enforce a PDE-consistent coupling.
- **Periodic Boundaries:** Similar penalties are applied at the domain boundaries to maintain global periodicity.

Summary:

- **Problem:** Tearing and blow-up occur because averaging alone does not enforce the PDE's flux consistency.
- **Cause:** Independent cell evolution without a PDE-based interface condition.
- **SAT Fix:** Adding SAT penalties at interfaces forces the cells to exchange proper flux information, eliminating discontinuities and preventing instability. However, we do not apply the SAT fix in this problem.

GitHub

GitHub link for the codes in the folder `Assignment 4`:
`https://github.com/nurmaton/SBP_KAUST/tree/main/Assignment%204`