Final Project of CFD Course: 038782
Lecturer: Prof.Steven Frankel

Ronith Stanly

Time spent: 20 Jan, 2019 to 21 Feb, 2019

From 1-D Scalar Advection to 2-D Compressible Navier-Stokes using SBP Operators, SAT Boundary Conditions and Split-form

Strand (1994); Carpenter et al. (1994); Kennedy and Gruber (2008)

1 Abstract

This study describes the descretization of several partial differential equations using fourth-order Summation By Parts (SBP) operators along with Simultaneous Approximation Term (SAT) boundary conditions. The ultimate goal of numerically solving the 2-D compressible Navier-Stokes equations is achieved by gradually progressing from a 1-D linear problems. The split-form fluxes are also employed in the descritization of the 2-D Compressible Navier-Stokes equations so as to study its effect on the stability of the solution process.

2 Introduction

There are several numerical methods to discretize a partial differential equation like finite difference, finite volume, finite element and flux reconstruction methods. In finite difference method, the domain of interest in discretized into a number of points on a grid and solution is computed at each of those points. Higher order can be achieved by employing stencils of varying width or by using compact stencils. Finite difference is done on cartesian or curvilinear grids and so has a restriction on the flexibility of the use of complex geometry, unless the Immersed Boundary Method (IBM) is used. A clever way to use the finite difference method is by using Summation By Parts (SBP) operators, which are essentially matrices with the coefficients used in the finite difference stencils with certain modification near the boundaries. The use of SBP operators are proven to be energy-stable for linear problems when used along with Simulatneous Approximation Term (SAT) boundary conditions. In finite volume method, the region of interest is decomposed into a number of cells and the solution is computed at cell centers. The flux reconstruction method is a clever combination of the finite volume and finite element methods and thereby offers the liberty of having a higher-order accurate solution by being able to compute the solution using multiple solution points within a cell, as opposed to computing solution just at the cell center in the finite volume method. This method also allows the ability to simulate flows over and through complex objects because it permits the use of unstructured grids. Recent studies (Fisher and Carpenter, 2013) have shown the equivalence between SBP operators and the flux reconstruction scheme implemented using Gauss-Lobatto points.

The following sections describe the implementation of the SBP-SAT method on several partial differential equations along with the equations, matrices, algorithms, implementation difficulties and their solutions. Section 3 describes the implementation of SBP operators to solve 1-d linear advection equation; Section 4 introduces the SAT term, Section 5 shows the implementation on a 1-D system of equations. Section 6 describes the application of SBP-SAT on 1-D Euler equations. Section 7 on 2-D linear advection and Section 7.1 describes the implementation of SATs in 2-D. Section 8 deals with 2-D Euler equations and the split-form of the fluxes are introduced in Section

8.2. Finally, the implementation on 2-D Compressible Navier-Stokes equations are described in Section 9.

3 SBP: 1-D Linear Scalar Advection

To check the implementaion of the SBP derivative operator, the 1D linear scalar advection equation was solved (without having any boundary conditions) and integrated upto different end times using RK-4.

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \tag{1}$$

$$f(u) = a \cdot u, \;\; a = 1 \tag{2}$$

$$\implies \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \implies \frac{\partial u}{\partial t} = -a\frac{\partial u}{\partial x} \tag{3}$$

For SBP, we have:

$$H \underbrace{D_1 \mathbf{u}}_{\frac{\partial \mathbf{u}}{\partial x}} = Q\mathbf{u} \tag{4}$$

where D_1 is the matrix derivative operator.

$$\implies \frac{\partial u}{\partial x} \approx H^{-1}Q\mathbf{u} \tag{5}$$

Applying SBP spatial discretization, Eq.(3) becomes,

$$\frac{\partial u}{\partial t} = -a \; H^{-1}Q\mathbf{u} \tag{6}$$

Initial condition:

$$u(x,0) = sin(\pi x) \tag{7}$$

Number of grid points, $N_x = 100$
CFL=0.1
The results are shown in figure 1.

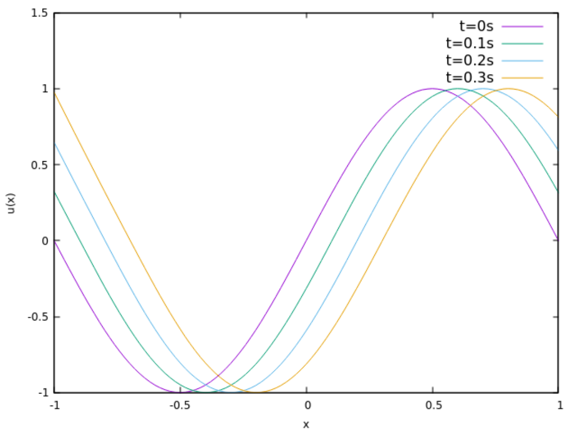


Figure 1: 1-D linear advection with SBP

!!! Check points:

1. Q matrix should look like what is given below with the lower right-hand looking as if it is the upper left hand part rotatedand given a negative sign.

$$Q = \begin{bmatrix} -0.5 & +0.6 & -0.8 & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & +0.8 & -0.6 & +0.5 \end{bmatrix} \tag{8}$$

2. And hence Q+Q^T should look like:

$$Q + Q^T = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 \end{bmatrix} \tag{9}$$

3. To check whether the derivative matrix is working well, take the derivative of x (i.e., grid points) and it should give a vector of ones:

$$D_1 \cdot x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{10}$$

4 SBP-SAT: 1-D linear scalar advection

Test case from the work of Carpenter et al. (1994) (pg.227):

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \ a = 1 \tag{11}$$

Domain:

$$0 \leq x \leq 1, \ t \geq 0 \tag{12}$$

Initial condition:

$$u(x, 0) = sin2\pi(x), \ 0 \leq x \leq 1 \tag{13}$$

Boundary condition (left boundary):

$$g(t) \equiv u(0, t) = sin2\pi(-t), \ t \geq 0 \tag{14}$$

Exact solution:

$$u(x, t) = sin2\pi(x - t) \tag{15}$$

$$\frac{\partial u}{\partial t} = -a\frac{\partial u}{\partial x} \tag{16}$$

Applying SBP spatial discretization with SAT boundary condition on left boundary, Eq.(16) becomes:

$$\frac{\partial \mathbf{u}}{\partial t} = \underbrace{-a\mathbf{H^{-1}Qu}}_{SBP} - \underbrace{\mathbf{H^{-1}}\sigma\left(u_{0,t} - g(t)\right)\mathbf{e_0}}_{SAT} \tag{17}$$

where $u_{0,t}$ is the solution value at the boundary at a given time; $g(t)$ is the target value at the boundary; energy estimate is non-negative for $\sigma \geq 0.5$ and here σ is taken as 0.5 (σ in the lecture 4, pg. 33 is related to Carpenter's τ as $\tau = 2\sigma$) and

$$\mathbf{e_0} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ . \\ . \\ 0 \end{bmatrix} \tag{18}$$

is the unit vector projected onto the first grid point if $a > 0$ (here since BC is applied on left boundary) otherwise projected on to last grid point:

$$\mathbf{e_N} = \begin{bmatrix} 0 \\ 0 \\ . \\ . \\ 0 \\ 1 \end{bmatrix} \tag{19}$$

Table 1: Errors in the 1D

Nx	L_2 **error**	$O(h)$
3	5.07992 E -4	–
6	7.98044 E -5	2.67
12	1.66426 E -5	2.26
24	2.33513 E -6	2.83
48	3.00401 E -7	2.95
96	3.78286 E -8	2.99

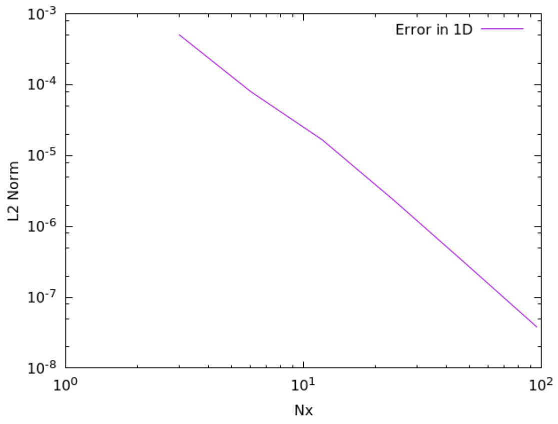


Figure 3: Convergence study in 1-D

The method used here should give fourth order accuracy in the interior and second order accuracy near the boundaries, such that it gives a global order of three. The results of the computed test case using 100 grid points is shown in figure 2.

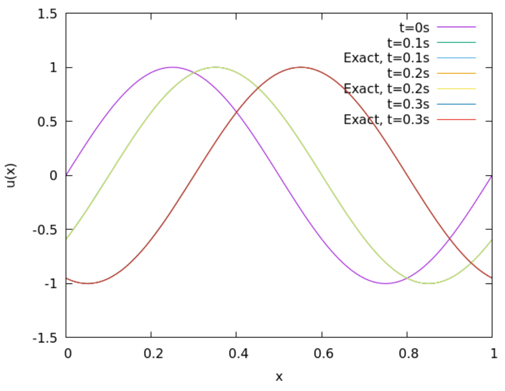


Figure 2: 1D linear advection using SBP and SAT boundary condition on the left boundary

The order of convergence of the solver in 1-D was checked using the same test-case but with peiodic boundary conditions imposed using SAT terms. Domain chosen haD the same size $0 \leq x \leq 1$ and was run for 20 time steps using a very small time step size 1×10^{-5}. The errors are as shown in Table 1 shown below. For the fourth-order SBP operators used here with periodic boundary conditions, a third-order convergence is expected. The results indicate a trend in the right direction.

5 SBP-SAT: 1-D system of equations

To move a step closer to the Navier-Stokes equation, another test case from Carpenter et al. (1994) (pg.228) involving system of equations (but still in 1-D) was computed.

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0 \tag{20}$$

$$\frac{\partial v}{\partial t} - \frac{\partial v}{\partial x} = 0 \tag{21}$$

Say:

$$u = u_1, \ v = u_2 \tag{22}$$

Domain:

$$0 \leq x \leq 1, \ t \geq 0 \tag{23}$$

Initial condition:

$$u(x,0) \equiv u_1(x,0) = sin2\pi(x), \ 0 \leq x \leq 1 \tag{24}$$

$$v(x,0) \equiv u_2(x,0) = -sin2\pi(x),\ 0 \le x \le 1 \tag{25}$$

Boundary condition (left boundary in first equation):

$$g_{1,left}(t) = \alpha u_2(0,t),\ t \ge 0 \tag{26}$$

Boundary condition (left boundary in second equation):

$$g_{2,left}(t) = 0 \tag{27}$$

Boundary condition (right boundary in first equation):

$$g_{1,right}(t) = 0 \tag{28}$$

Boundary condition (right boundary in second equation):

$$g_{2,right}(t) = \beta u_1(N,t),\ t \ge 0 \tag{29}$$

So as to be able to compare with an exact solution, taking $\alpha = \beta = 1$.

Exact solutions for $\alpha = \beta = 1$:

$$u_1(x,t) \equiv u(x,t) = sin2\pi(x-t) \tag{30}$$

$$u_2(x,t) \equiv v(x,t) = -sin2\pi(x+t) \tag{31}$$

Writing equations (20) and (21) in the vector/CFD form:

$$\frac{\partial}{\partial t}\begin{bmatrix}\mathbf{u_1}\\ \mathbf{u_2}\end{bmatrix} = -\frac{\partial}{\partial x}\begin{bmatrix}\mathbf{f_1}\\ \mathbf{f_2}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,left}}\\ \mathbf{SAT_{2,left}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,right}}\\ \mathbf{SAT_{2,right}}\end{bmatrix} \tag{32}$$

where:

$$\mathbf{u_1} = \mathbf{u},\ \mathbf{u_2} = \mathbf{v} \tag{33}$$

$$\mathbf{f_1} = +\mathbf{u_1},\ \mathbf{f_2} = -\mathbf{u_2} \tag{34}$$

$$\mathbf{SAT_{1,left}} = \mathbf{H^{-1}}\sigma\left(u_1(0,t) - g_{1,left}(t)\right)\mathbf{e_0}, \tag{35}$$

$$\mathbf{SAT_{2,left}} = \mathbf{H^{-1}}\sigma\left(u_2(0,t) - g_{2,left}(t)\right)\mathbf{e_0} \tag{36}$$

$$\mathbf{SAT_{1,right}} = \mathbf{H^{-1}}\sigma\left(u_1(N,t) - g_{1,right}(t)\right)\mathbf{e_N}, \tag{37}$$

$$\mathbf{SAT_{2,right}} = \mathbf{H^{-1}}\sigma\left(u_2(N,t) - g_{2,right}(t)\right)\mathbf{e_N} \tag{38}$$

The results of the computed test case using 100 grid points is shown in figure 4.

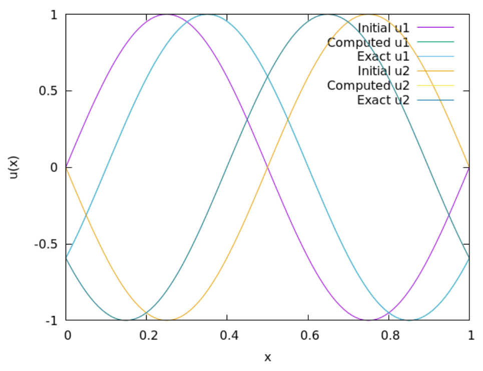


Figure 4: 1-D linear advection using SBP and SAT on a system of equations

6 SBP-SAT: 1-D Euler equations

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho \\ \rho u \\ \rho\left(\frac{1}{2}u^2+e\right)\end{bmatrix} = -\frac{\partial}{\partial x}\begin{bmatrix}\rho u \\ \rho u^2+p \\ u\rho\left(\frac{1}{2}u^2+e\right)+up\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,left}} \\ \mathbf{SAT_{2,left}} \\ \mathbf{SAT_{3,left}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,right}} \\ \mathbf{SAT_{2,right}} \\ \mathbf{SAT_{3,right}}\end{bmatrix} \tag{39}$$

where:

$$e = \frac{p}{(\gamma-1)\rho} \tag{40}$$

$$\underbrace{\frac{\partial}{\partial t}\begin{bmatrix}\mathbf{u_1} \\ \mathbf{u_2} \\ \mathbf{u_3}\end{bmatrix}}_{Solution\ Vector} = -\underbrace{\frac{\partial}{\partial x}\begin{bmatrix}\mathbf{f_1} \\ \mathbf{f_2} \\ \mathbf{f_3}\end{bmatrix}}_{Flux\ Vector} - \underbrace{\begin{bmatrix}\mathbf{H^{-1}}\sigma\left(u_1(0,t)-g_{1,left}(t)\right)\mathbf{e_0} \\ \mathbf{H^{-1}}\sigma\left(u_2(0,t)-g_{2,left}(t)\right)\mathbf{e_0} \\ \mathbf{H^{-1}}\sigma\left(u_3(0,t)-g_{3,left}(t)\right)\mathbf{e_0}\end{bmatrix}}_{SAT_{Left}} - \underbrace{\begin{bmatrix}\mathbf{H^{-1}}\sigma\left(u_1(N,t)-g_{1,right}(t)\right)\mathbf{e_N} \\ \mathbf{H^{-1}}\sigma\left(u_2(N,t)-g_{2,right}(t)\right)\mathbf{e_N} \\ \mathbf{H^{-1}}\sigma\left(u_3(N,t)-g_{3,right}(t)\right)\mathbf{e_N}\end{bmatrix}}_{SAT_{Right}} \tag{41}$$

6.1 Algorithm:

1. Initialize primitive variables: ρ, u, p

   ```
   CALL inputs
   CALL grid1d
   CALL init1d_shock_tube
   ```

2. Construct solution vector from the premitive variables

   ```
   CALL solvec
   ```

3. Time loop:

   (a) Construct flux vector from the solution vector

   ```
   CALL flux
   ```

   (b) Compute derivative of flux vector

   ```
   CALL dflux_dx
   ```

   (c) Compute SAT terms

   ```
   CALL SAT_x_left
   CALL SAT_x_right
   ```

   (d) Combine the above two to get RHS

   ```
   CALL right_hand_side
   ```

   (e) Integrate in time

   ```
   CALL rk4
   ```

4. At the end of each RK-4 stage (after the solution vector has been computed), decompose the solution vector to get back primitive variables. This will provide the pressure term which is needed to compute flux in the next RK-4 stage and next time step.

   ```
   CALL decomp
   ```

6.2 Problem Description: Sod's shock-tube

The usual shock-tube problem has initially discontinuous conditions for density and pressure as given below:

$$\begin{bmatrix}\rho \\ u \\ p\end{bmatrix}_{t=0} = \begin{cases}\begin{bmatrix}1.0 \\ 0.0 \\ 1.0\end{bmatrix}, & \text{if } 0 < x \leq 0.5 \\ \begin{bmatrix}0.125 \\ 0.0 \\ 0.1\end{bmatrix}, & \text{if } 0.5 < x \leq 1\end{cases}$$

Since the scheme used here is essential a centered difference scheme, the solution blew-up when this discontinuous intial conditions were inputted. Hence a hyperbolic tangent was used, as shown below, to smoothen the initial conditions a bit.

```
rho = (rhol+rhor)*0.5 + (rhol-rhor)*0.5*tanh(200.*(xd-x))
pre = (prel+prer)*0.5 + (prel-prer)*0.5*tanh(200.*(xd-x))
```

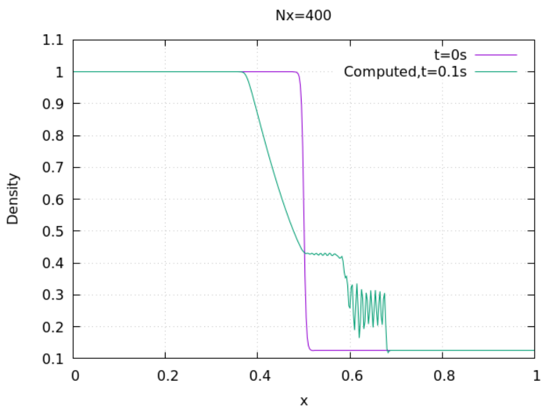
Dirichlet boundary conditions were used; wherein the conserved quantities take on the values specified by the initial conditions at either boundary.

$$g_{nvar,\ left}(t) \equiv u_{nvar}(0,t) = u_{nvar}(0,0),\ \text{where } nvar \in (1,2,3) \tag{42}$$

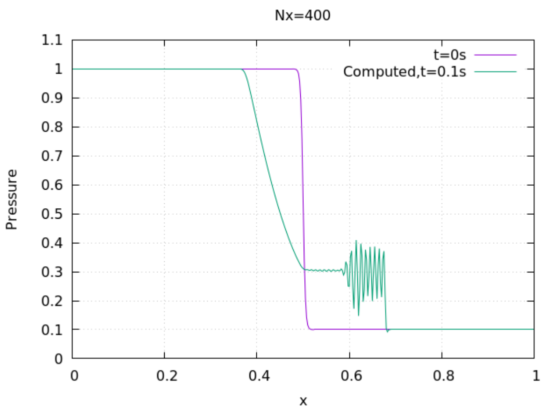
$$g_{nvar,\ right}(t) \equiv u_{nvar}(N,t) = u_{nvar}(N,0),\ \text{where } nvar \in (1,2,3) \tag{43}$$

6.2.1 Results

The results shown in Figures 5a and 5b indicate that even though there are oscillations around the region of shock (which is due to the usage of central scheme and also because no artificial dissipation was used), the implementation of the algorithm for the 1-D Euler equations is proper.



(a) Density



(b) Pressure

Figure 5: Sod's shock-tube problem

7 SBP-SAT: 2-D Linear Scalar Advection

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y} = 0 \tag{44}$$

2-D Domain: $-3 \leq x < 3$, $-3 \leq y < 3$
Initial Gaussian profile:

$$u(x,y,0) \equiv u_0(x,y) = \exp\left[-\left(\frac{x^2}{2} + \frac{y^2}{2}\right)\right] \tag{45}$$

Exact solution at time, t:

$$u(x,y,t) = u_0(x-at, y-bt) \equiv \exp\left[-\left(\frac{(x-at)^2}{2} + \frac{(y-bt)^2}{2}\right)\right] \tag{46}$$

With periodic SAT boundary conditions:

$$\frac{\partial u}{\partial t} = -a\frac{\partial u}{\partial x} - b\frac{\partial u}{\partial y} \tag{47}$$

$$\frac{\partial \mathbf{u}}{\partial t} = \underbrace{-a\mathbf{H_x^{-1}Qu}}_{SBP_X} - \underbrace{-b\mathbf{H_y^{-1}Qu}}_{SBP_Y} - \underbrace{\mathbf{H_x^{-1}}\sigma\left(u_{0,y,t} - u_{Nx,y,t}\right)\mathbf{e_0}}_{SAT_{X_{Left}}} - \underbrace{\mathbf{H_x^{-1}}\sigma\left(u_{Nx,y,t} - u_{0,y,t}\right)\mathbf{e_N}}_{SAT_{X_{Right}}} \tag{48}$$

$$- \underbrace{\mathbf{H_y^{-1}}\sigma\left(u_{x,0,t} - u_{x,Ny,t}\right)\mathbf{e_0}}_{SAT_{Y_{Bottom}}} - \underbrace{\mathbf{H_y^{-1}}\sigma\left(u_{x,Ny,t} - u_{x,0,t}\right)\mathbf{e_N}}_{SAT_{Y_{Top}}} \tag{49}$$

Table 2: Errors in 2-D linear scalar advection

Nx=Ny	L_2 **error**	$O(h)$
03	9.02300 E -5	—
06	8.23020 E -6	3.45
12	2.51993 E -6	1.70
24	3.73166 E -7	2.75
48	5.85714 E -8	2.67

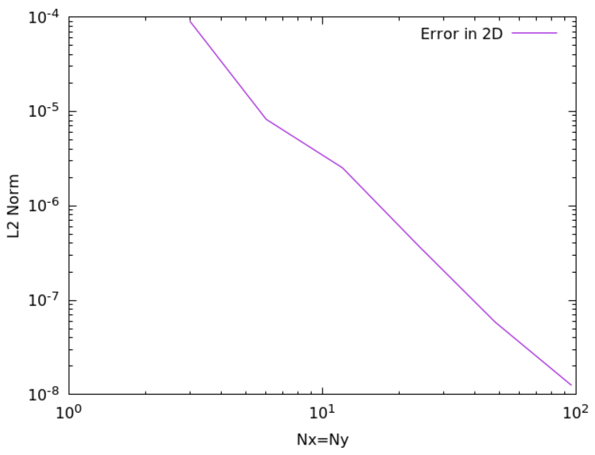
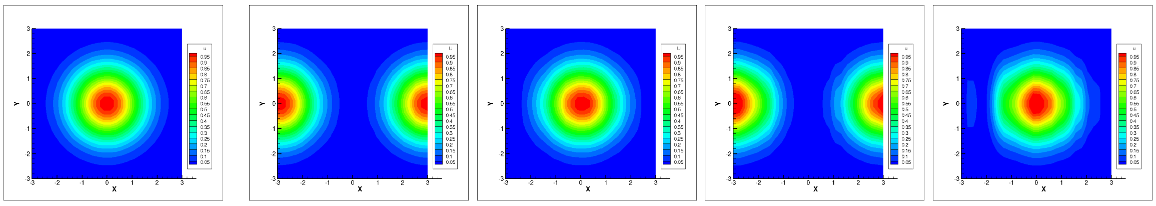


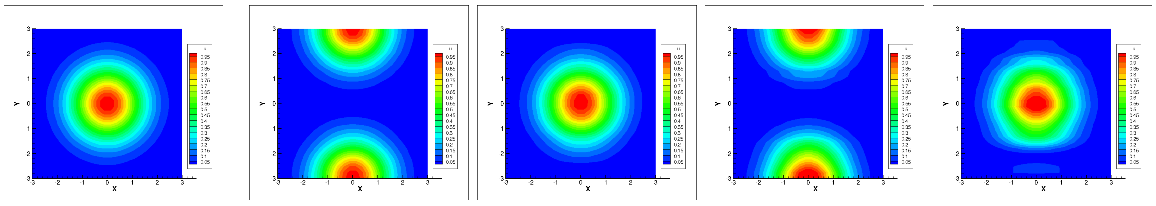
Figure 8: Convergence study in 2-D linear scalar advection

The results for two cycles with $N_x = N_y = 25$ and $CFL = 0.1$ are shown in the figures 6 and 7; with the initial Gaussian profile moving from right to left in figure 6 and from bottom to top in figure 7. When it was ran for four cycles and more, the maximum and minimum value of u kept increasing. These oscillations might be because of the central scheme that was used and due to the lack of any dissipative terms.



(a) t=0s (b) t=3s (c) t=6s (d) t=9s (e) t=12s

Figure 6: Velocities: a=1, b=0



(a) t=0s (b) t=3s (c) t=6s (d) t=9s (e) t=12s

Figure 7: Velocities: a=0, b=1

The order of convergence of the solver in 2-D was checked using this test-case on a domain of size $-3 \le x \le 3$, $-3 \le y \le 3$ by running it for 20 time steps using a very small time step size 1×10^{-5}. The errors are plotted in Figure 8 and are also shown in Table 2. For the fourth-order SBP operators used here with periodic boudnary conditions, a thrid-order convergence is expected. The results indicate a trend in the right direction.

7.1 SATs in 2-D

Applying the SAT boundary conditions in 2-D can be quite confusing, since most of the research articles make use of Kronecker products in their equations (for mathematical analysis), but at the same time, they state that the implementation in a code does not use the same strategy. Even though it can be implemented in the same way (i.e., by using Kronecker products) in a code, that is not the most efficient way to do it; not just because such matrix operations are expensive, but it also requires some post-processing of the resulting matrix to comply with the ranks of the other arrays on the right-hand side of the equation (after all, the matrix is composed mostly of zeros which itself motivates us to not use the costly matrix multiplication on a bunch of zeros just to give back zeros). This is because the outputted quantity (after taking Kronecker product) is a huge 1-D vector rather than a 2-D vector (i.e., matrix) as exhibited below:

$$\mathbf{A} = (a_{ij})\ ,\ \mathbf{B} = (b_{ij}) \tag{50}$$

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & .. & a_{1n-1}\mathbf{B} & a_{1n}\mathbf{B} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & .. & a_{mn-1}\mathbf{B} & a_{mn}\mathbf{B} \end{bmatrix} \tag{51}$$

In 1-D, we get a vector of size $(1 \times nx)$:

$$\sigma \quad \underbrace{\mathbf{H}_x^{-1}}_{Matrix\ (nx \times nx)} \quad \underbrace{\mathbf{e}_{x=0}}_{Vector\ (1 \times nx)} \quad (u_0 - g) \tag{52}$$

In 2-D, after taking Kronecker product, we get a vector of size $(1 \times nxny)$:

Equations (48) and (49) can be written using Kronecker products (simplified from Lundquist and Nordström (2016) [Eq.6]):

$$\mathbf{u}_t + \mathbf{u}_x + \mathbf{u}_y = 0 \tag{53}$$

$$\mathbf{u}_t + \underbrace{\left(\mathbf{D}_x^{-1} \otimes \mathbf{I}_y\right)\mathbf{u}}_{\mathbf{u}_x} + \underbrace{\left(\mathbf{I}_x \otimes \mathbf{D}_y\right)\mathbf{u}}_{\mathbf{u}_y} = \underbrace{\sigma_{x=0}\left(\mathbf{H}_x^{-1}\mathbf{e}_{x=0} \otimes \mathbf{u}|_{x=0} - g(t,y)\right)}_{SAT_{Left}\&\ SAT_{Right}} + \underbrace{\sigma_{y=0}\left(\mathbf{u}|_{y=0} - g(t,x) \otimes \mathbf{H}_y^{-1}\mathbf{e}_{y=0}\right)}_{SAT_{Top}\&\ SAT_{Bottom}} \tag{54}$$

where:

$$\sigma_{x=0}\left(\underbrace{\mathbf{H}_x^{-1}}_{Matrix\ (nx \times nx)} \otimes \underbrace{\mathbf{H}_y^{-1}}_{Matrix\ (ny \times ny)}\right) \tag{55}$$

$$\equiv \sigma_{x=0}\left(\mathbf{H}_x^{-1} \otimes \mathbf{I}_y\right)\left(\mathbf{e}_{x=0} \otimes \mathbf{u}|_{x=0} - g(t,y)\right) \tag{56}$$

$$\underbrace{\equiv \sigma_{x=0}\left(\underbrace{\mathbf{H}_x^{-1}}_{Matrix\ (nx \times nx)} \underbrace{\mathbf{e}_{x=0}}_{Vector\ (nx)} \otimes \underbrace{\mathbf{u}|_{x=0}}_{Vector\ (1 \times ny)} - g(t,y)\right)}_{Vector\ (1 \times nxny)} \tag{57}$$

In order for this $(1 \times nxny)$ vector to be compatible with the other terms on the right-hand side (like matrices of size $(nx \times ny)$ for terms including $-dfdx$, $-dgdy$) this vector should be restructured (which is difficult).

There are two ways to get around this problem: one is to carefully implement the 1-D method in 2-D (by taking each x-point and considering the entire set of points in y-direction at that x-location as a 1-D vector; and then moving along all the x-points in the same manner). This is implemented as shown in equations (48) and (49). The second method is the most computationally efficient one as it excludes the use of Kronecker products and associated matrix operations by carefully appending desired boundary values to a matrix initiated with zeros. This is shown below:

For SAT_{Left}:
Let $h_{x0} = \mathbf{H}_x^{-1}(0,0)$

$$SAT_{Left}\ (nx \times ny) \equiv SAT_{x0} = \sigma_{x0} \cdot \begin{bmatrix} (u_{x=0,y=0} - g(t,y=0)) \cdot h_{x0} & 0 & 0 & ... & 0 & 0_{nx} \\ (u_{x=0,y=y} - g(t,y=\Delta y)) \cdot h_{x0} & 0 & 0 & ... & 0 & 0_{nx} \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ (u_{x=0,y=ny} - g(t,y=ny)) \cdot h_{x0} & 0 & 0 & ... & 0 & 0_{nx} \end{bmatrix} \tag{58}$$

For SAT_{Right}:
Let $h_{xn} = \mathbf{H}_x^{-1}(nx, nx)$

$$SAT_{Right}\ (nx \times ny) \equiv SAT_{xn} = \sigma_{xn} \cdot \begin{bmatrix} 0 & 0 & 0 & ... & 0 & (u_{x=x_n,y=0} - g(t,y=0)) \cdot h_{x_n} \\ 0 & 0 & 0 & ... & 0 & (u_{x=x_n,y=\Delta y} - g(t,y=\Delta y)) \cdot h_{x_n} \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ 0 & 0 & 0 & ... & 0 & (u_{x=x_n,y=ny} - g(t,y=ny)) \cdot h_{x_n} \end{bmatrix} \tag{59}$$

For SAT_{Top}:
Let $h_{y0} = \mathbf{H}_y^{-1}(0, 0)$

$$a_1 = (u_{x=0,y=0} - g(t,y=0)) \cdot h_{y0} \tag{60}$$

$$a_2 = (u_{x=\Delta x,y=y} - g(t,y=\Delta y)) \cdot h_{y0} \tag{61}$$

$$a_{nx-1} = (u_{x=nx-1,y=ny} - g(t,y=ny)) \cdot h_{y0} \tag{62}$$

$$a_{nx} = (u_{x=nx,y=ny} - g(t,y=ny)) \cdot h_{y0} \tag{63}$$

$$SAT_{Top}\ (nx \times ny) \equiv SAT_{y0} = \sigma_{y0} \cdot \begin{bmatrix} a_1 & a_2 & . & ... & a_{nx-1} & a_{nx} \\ 0 & 0 & 0 & ... & 0 & 0_{nx} \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ 0 & 0 & 0 & ... & 0 & 0_{nx} \end{bmatrix} \tag{64}$$

For SAT_{Bottom}:
Let $h_{yn} = \mathbf{H}_y^{-1}(ny, ny)$

$$a_1 = (u_{x=0,y=ny} - g(t,y=ny)) \cdot h_{yn} \tag{65}$$

$$a_2 = (u_{x=\Delta x,y=ny} - g(t,y=ny)) \cdot h_{yn} \tag{66}$$

$$a_{nx-1} = (u_{x=nx-1,y=ny} - g(t,y=ny)) \cdot h_{yn} \tag{67}$$

$$a_{nx} = (u_{x=nx,y=ny} - g(t,y=ny)) \cdot h_{yn} \tag{68}$$

$$SAT_{Bottom}\ (nx \times ny) \equiv SAT_{yn} = \sigma_{yn} \cdot \begin{bmatrix} 0 & 0 & 0 & ... & 0 & 0_{nx} \\ 0 & 0 & 0 & ... & 0 & 0_{nx} \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ 0 & 0 & 0 & ... & 0 & 0_{nx} \\ a_1 & a_2 & . & ... & a_{nx-1} & a_{nx} \end{bmatrix} \tag{69}$$

8 SBP-SAT: 2-D Euler Equations

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho\left(\frac{1}{2}(u^2+v^2)+e\right)\end{bmatrix} = -\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)+e\right)+up\end{bmatrix} - \frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ v\rho\left(\frac{1}{2}(u^2+v^2)+e\right)+vp\end{bmatrix} \tag{70}$$

$$-\begin{bmatrix}\mathbf{SAT_{1,left}}\\ \mathbf{SAT_{2,left}}\\ \mathbf{SAT_{3,left}}\\ \mathbf{SAT_{4,left}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,right}}\\ \mathbf{SAT_{2,right}}\\ \mathbf{SAT_{3,right}}\\ \mathbf{SAT_{4,right}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,bottom}}\\ \mathbf{SAT_{2,bottom}}\\ \mathbf{SAT_{3,bottom}}\\ \mathbf{SAT_{3,bottom}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,top}}\\ \mathbf{SAT_{2,top}}\\ \mathbf{SAT_{3,top}}\\ \mathbf{SAT_{4,top}}\end{bmatrix} \tag{71}$$

where:

$$e = \frac{p}{(\gamma-1)\rho} \tag{72}$$

Discretizing using SBP operators and weakly imposing periodic boundary conditions using SAT terms:

$$\underbrace{\frac{\partial}{\partial t}\begin{bmatrix}\mathbf{u_1}\\ \mathbf{u_2}\\ \mathbf{u_3}\\ \mathbf{u_4}\end{bmatrix}}_{Solution\ Vector} = \underbrace{-\mathbf{H_x^{-1}Q}\begin{bmatrix}\mathbf{f_1}\\ \mathbf{f_2}\\ \mathbf{f_3}\\ \mathbf{f_4}\end{bmatrix} - \mathbf{H_y^{-1}Q}\begin{bmatrix}\mathbf{g_1}\\ \mathbf{g_2}\\ \mathbf{g_3}\\ \mathbf{g_4}\end{bmatrix}}_{Flux\ Vectors} - \underbrace{\begin{bmatrix}\mathbf{H_x^{-1}}\sigma\left(u_1(0,y,t)-u_1(N,y,t)\right)\mathbf{e_{x,0}}\\ \mathbf{H_x^{-1}}\sigma\left(u_2(0,y,t)-u_2(N,y,t)\right)\mathbf{e_{x,0}}\\ \mathbf{H_x^{-1}}\sigma\left(u_3(0,y,t)-u_3(N,y,t)\right)\mathbf{e_{x,0}}\\ \mathbf{H_x^{-1}}\sigma\left(u_4(0,y,t)-u_4(N,y,t)\right)\mathbf{e_{x,0}}\end{bmatrix}}_{SAT_{X\,Left}} \tag{73}$$

$$-\underbrace{\begin{bmatrix}\mathbf{H_x^{-1}}\sigma\left(u_1(N,y,t)-u_1(0,y,t)\right)\mathbf{e_{x,n}}\\ \mathbf{H_x^{-1}}\sigma\left(u_2(N,y,t)-u_2(0,y,t)\right)\mathbf{e_{x,n}}\\ \mathbf{H_x^{-1}}\sigma\left(u_3(N,y,t)-u_3(0,y,t)\right)\mathbf{e_{x,n}}\\ \mathbf{H_x^{-1}}\sigma\left(u_4(N,y,t)-u_4(0,y,t)\right)\mathbf{e_{x,n}}\end{bmatrix}}_{SAT_{X\,Right}} \tag{74}$$

$$-\underbrace{\begin{bmatrix}\mathbf{H_y^{-1}}\sigma\left(u_1(x,0,t)-u_1(x,N,t)\right)\mathbf{e_{y,0}}\\ \mathbf{H_y^{-1}}\sigma\left(u_2(x,0,t)-u_2(x,N,t)\right)\mathbf{e_{y,0}}\\ \mathbf{H_y^{-1}}\sigma\left(u_3(x,0,t)-u_3(x,N,t)\right)\mathbf{e_{y,0}}\\ \mathbf{H_y^{-1}}\sigma\left(u_4(x,0,t)-u_4(x,N,t)\right)\mathbf{e_{y,0}}\end{bmatrix}}_{SAT_{Y\,Bottom}} - \underbrace{\begin{bmatrix}\mathbf{H_y^{-1}}\sigma\left(u_1(x,N,t)-u_1(x,0,t)\right)\mathbf{e_{y,n}}\\ \mathbf{H_y^{-1}}\sigma\left(u_2(x,N,t)-u_2(x,0,t)\right)\mathbf{e_{y,n}}\\ \mathbf{H_y^{-1}}\sigma\left(u_3(x,N,t)-u_3(x,0,t)\right)\mathbf{e_{y,n}}\\ \mathbf{H_y^{-1}}\sigma\left(u_4(x,N,t)-u_4(x,0,t)\right)\mathbf{e_{y,n}}\end{bmatrix}}_{SAT_{Y\,Top}} \tag{75}$$

8.1 Problem Description: Isentropic Vortex

Test case of isentropic vortex proposed in the work of Yee et al. (2000)(pg.59) is written below. Its analytical solution at time, t, was computed using the equation in Fisher and Carpenter (2013)(pg.541):
Initial conditions:

$$\rho = \left[1 - \frac{\beta^2(\gamma-1)}{8\gamma\pi^2}\exp\left(1-r^2\right)\right]^{\frac{1}{\gamma-1}} \tag{76}$$

where radius of vortex:

$$r = \sqrt{(x-x_c)^2 + (y-y_c)^2} \tag{77}$$

$$u = M\cos\alpha - \frac{\beta(y-y_c)}{2\pi}\exp\left(\frac{1-r^2}{2}\right) \tag{78}$$

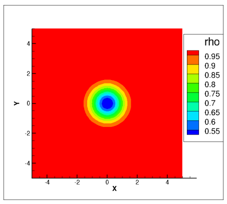
$$v = M\sin\alpha - \frac{\beta(x-x_c)}{2\pi}\exp\left(\frac{1-r^2}{2}\right) \tag{79}$$

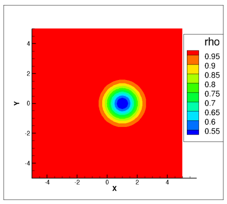
$$p = \rho^\gamma \tag{80}$$

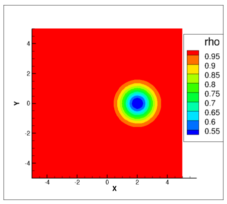
where, center of vortex is given as $x_c = 0,\ y_c = 0$; the advection angle $\alpha = 0°$; the vortex strength, $\beta = 5$ and the mach number, $M = 1$. To maintain isentropic condition in the flow-field, the decompose routine should also update pressure in the manner shown above.

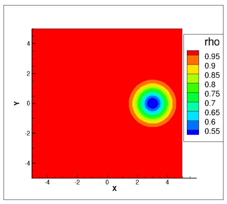
8.1.1 Results and Discussion

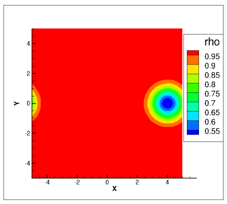
The results obtained by using $Nx = Ny = 50$, $CFL = 0.1$ and $\sigma = 0.5$ are shown in Figure 9 and using $\sigma = 4$ in Figure 10. Since the SBP operator used here is based on a centered scheme, it produces oscillations in the results. These oscillations grew with time and with grid refinement. The lack of any viscous terms might have encouraged the growth of these spurious oscillations which then destabilized the simulation before the vortex completed one full cycle. It is well known that even in the absence of discontinuities like shockwaves, a less (or none) dissipative method is generally unstable due to the accumulation of aliasing errors caused by discretized non-linear convective terms (Abe et al., 2018). But the value of σ used in the SAT term seemed to have an effect in stabilizing the simulation (upto a certain limit). With a $\sigma = 0.5$, the solution diverged before the isentropic vortex completed one cycle at $t = 10s$ (Figure 9), but with a $\sigma = 4$, the simulation stayed stable for upto $t = 30s$ (i.e., 3 cycles), as shown in Figure10. However, usually the value of σ in the SAT term is cleverly constructed (or in some cases dynamically adjusted) just to ensure that the required boundary conditions are established in shortest number of time-steps.
The results obtained here (i.e., the number of cycles completed by the isentropic vortex) is similar to what was obtained by some other people like Yee et al. (2000).

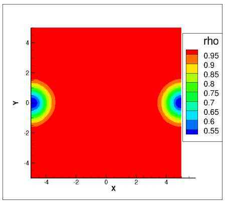

(a) t=0s

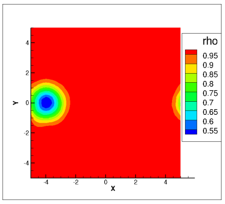

(b) t=1s

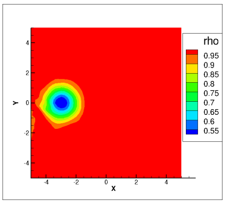

(c) t=2s

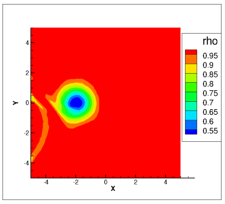

(d) t=3s


(e) t=4s


(f) t=5s


(g) t=6s


(h) t=7s


(i) t=8s

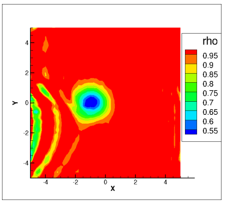
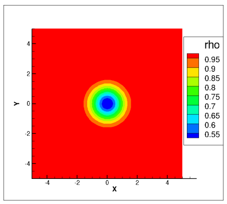
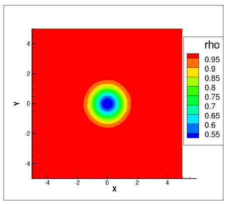
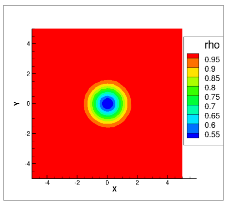
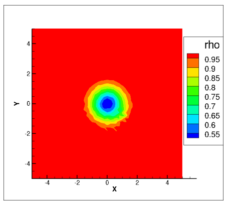

(j) t=9s

Figure 9: Isentropic Vortex using $\sigma = 0.5$ in 2-D Euler code


(a) t=0s


(b) t=10s


(c) t=20s


(d) t=30s

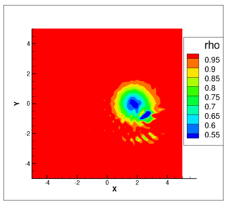

(e) t=31.7s

Figure 10: Isentropic Vortex using $\sigma = 4.0$ in 2-D Euler code

8.2 Split-form and stability of Euler equations

Even though SBP-SAT method is "energy stable", it's energy stability has been proven, and is valid, only for linear problems (Crean et al., 2017). This means that the "energy stable" SBP-SAT method may not be stable for non-linear systems of equations like the compressible Navier-Stokes equations without additional treatments like artificial dissipation, upwinding, etc. There are some ways that are actively being studied to make the scheme more stable for complex equations, like the use of flux splitting or even using entropy stable formulations; but these are usually harder to implement than the ones mentioned before. Entropy stable formulations are designed to be conditionally stable for non-linear problems. They make use of the concept that, provided density and pressure remain positive, entropy becomes a convex combination (remains positive) of momentum, density and energy; thereby ensuring an estimate for the entropy (i.e., entropy does not grow to ∞) and in-turn for momentum, density and pressure. But numerically, pressure and density may not always remain positive (for example, consider the expansion of gas into vacuum where the pressure jumps from a given value to zero which produces a shock and the numerical oscillations near the shock, especially the ones near zero will become negative) and hence specific treatment is required, like limiters, to simulate such problems even with the entropy-stable formulations.

In this study, we investigate one of the de-aliasing strategies used in high-order communities such as finite differences. A very prominent example for de-aliasing is the use of alternative formulations of the non-linear advection terms, e.g. so-called split formulations.(Gassner et al., 2016). Here, a variant of the energy-preserving flux-splitting method introduced by Kennedy and Gruber (2008) and referred to as $Split_{KG1}$ in Abe et al. (2018) was implemented to check whether it made the solution process more stable. Abe et al. (2018) introduced a two-part split-form, where the definition of a quadratic split-forms was extended in such a way that it comprised of a cubic non-linear term for ρ,$\mathbf{u}$,and ϕ, but it's splitting structure was a combination of three different quadratic split-forms ($\rho\mathbf{u}\phi \to \rho\mathbf{u}\cdot\phi$, $\rho\mathbf{u}\phi \to \rho\phi\cdot\mathbf{u}$ and $\rho\mathbf{u}\phi \to \mathbf{u}\phi\cdot\rho$).

The equations shown in Abe et al. (2018)[Eq.29 and Eq.32] were extended to 2-D and the convective terms in x and y directions were split as shown below. The pressure term is separated out from the convective term, because the flux-splitting does not act on them.
The normal or usual form (hereafter referred to as the un-split form) given in Eq.(71) is re-stated below:

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho\left(\frac{1}{2}(u^2+v^2)+e\right)\end{bmatrix} = -\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)+e\right)+up\end{bmatrix} - \frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ v\rho\left(\frac{1}{2}(u^2+v^2)+e\right)+vp\end{bmatrix} \tag{81}$$

where $e=\frac{p}{(\gamma-1)\rho}$

From the above pressure term can be taken out as:

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho\left(\frac{1}{2}(u^2+v^2)+e\right)\end{bmatrix} = -\underbrace{\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{Convective\ Term\ 1} - \frac{\partial}{\partial x}\begin{bmatrix}0\\ p\\ 0\\ \frac{up}{\gamma-1}+up\end{bmatrix} \tag{82}$$

$$-\underbrace{\frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2\\ v\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{Convective\ Term\ 2} - \frac{\partial}{\partial y}\begin{bmatrix}0\\ 0\\ p\\ \frac{vp}{\gamma-1}+vp\end{bmatrix} \tag{83}$$

The first convective term can be split as:

(the variables used for the underbrace are the same as used in the code)

$$\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix} \equiv \alpha\cdot\underbrace{\frac{\partial}{\partial x}\underbrace{\left\{\rho u\underbrace{\begin{bmatrix}1\\ u\\ v\\ \frac{1}{2}(u^2+v^2)\end{bmatrix}}_{\phi}\right\}}_{f_1}}_{Split\ x_1} \tag{84}$$

$$+\beta\cdot\left[\underbrace{\underbrace{\left\{\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}\right\}}_{\phi}\cdot\frac{\partial}{\partial x}\cdot\underbrace{(\rho u)}_{f_2}}_{Split\ x_2}+\underbrace{u\cdot\frac{\partial}{\partial x}\underbrace{\left\{\rho\cdot\underbrace{\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{\phi}\right\}}_{f_3}}_{Split\ x_3}+\underbrace{\rho\cdot\frac{\partial}{\partial x}\cdot\underbrace{\left\{u\cdot\underbrace{\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{\phi}\right\}}_{f_4}}_{Split\ x_4}\right] \tag{85}$$

$$+\left[\underbrace{(\rho u)\cdot\frac{\partial}{\partial x}\cdot\underbrace{\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{\phi=f_5}}_{Split\ x_5}+\underbrace{\left\{\rho\cdot\underbrace{\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{\phi}\right\}\cdot\frac{\partial}{\partial x}\underbrace{u}_{f_6}}_{Split\ x_6}+\underbrace{\left\{u\cdot\underbrace{\begin{bmatrix}\rho u\\ \rho u^2\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)\right)\end{bmatrix}}_{\phi}\right\}\cdot\frac{\partial}{\partial x}\underbrace{\rho}_{f_7}}_{Split\ x_7}\right] \tag{86}$$

$$\cdot(1-\alpha-2\beta) \tag{87}$$

For the second convective term (term in y), the ϕ matrix remained the same, the only changes made to the above were to change $\frac{\partial}{\partial x}\rightarrow\frac{\partial}{\partial y}$ and $u\rightarrow v$. The values of α and β were chosen as $\alpha=\beta=\frac{1}{4}$, as used by Abe et al. (2018).
The same isentropic vortex was simulated using the above split-form and the solution stayed stable for double the time (≈ 6 cycles). The results are shown in Figure 11. In order to investigate further and to understand why the un-split formulation diverged earlier, the integral of kinetic energy and entropy over the entire domain was studied as shown in Figure 12.

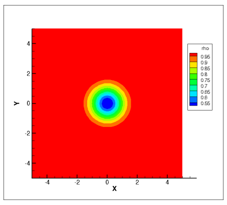
Kinetic energy was computed as:

$$KE=\int_{i=0}^{i=Nx}\int_{j=0}^{j=Ny}\frac{1}{2}\rho\left(u^2+v^2\right) \tag{88}$$

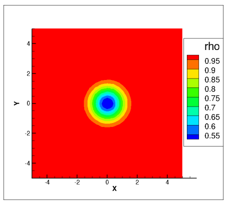
and entropy as:

$$S=\int_{i=0}^{i=Nx}\int_{j=0}^{j=Ny}\gamma log\left(\frac{p}{\rho}\right) \tag{89}$$

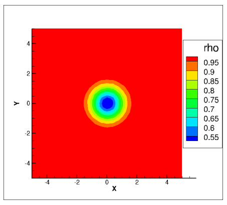
It can be seen that each time the vortex crossed the right boundary and entered back into the domain through the left boundary, the kinetic energy shoots-up and the entropy dips down dramatically. The jump in kinetic energy slowly increases the mean kinetic energy (seen from the overall slope of the curve) until it finally destabilizes the solution. The split formulation is better capable of sustaining the solution process for a longer time despite the jumps. As a side note, it was observed that the value of σ had an influence even with this split formulation. The solution diverged before the vortex completed one cycle with a $\sigma = 0.5$ (similar to what was observed in the un-split case); however, when a $\sigma = 4$ was used it gave the results mentioned above (i.e., double the number of cycles than the un-split case with the same value of σ).



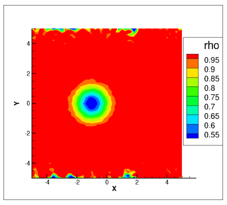
(a) t=0s



(b) t=20s

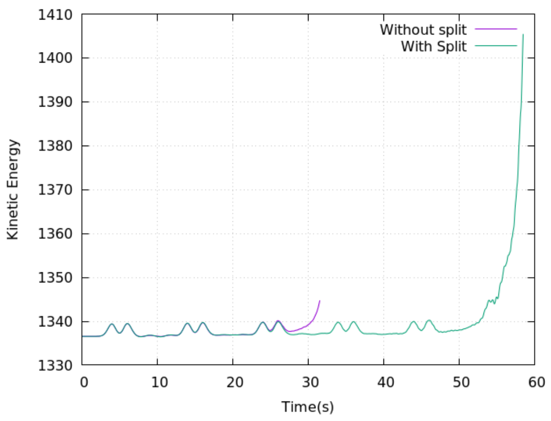


(c) t=40s

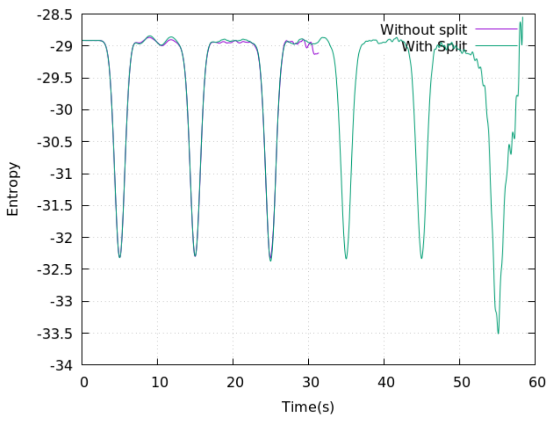


(d) t=58.9s

Figure 11: Isentropic Vortex using split form and $\sigma = 4.0$ in 2-D Euler code



(a) Kinetic Energy with time



(b) Entropy with time

Figure 12: Temporal statistics of Isentropic Euler Vortex

9 SBP-SAT: 2-D Compressible Navier-Stokes Equation

The 2-D compressible Navier-Stokes equation can be split into viscous and inviscid parts and can be written in the convenient vector form as given below. This allows for the easy conversion of this full compressible 2-D Navier-Stokes equations to the 2-D Euler equations by simply setting the viscous terms to zero.

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial \mathbf{x}}\left(\mathbf{F}_{invis} - \mathbf{F}_{visc}\right) + \frac{\partial}{\partial \mathbf{y}}\left(\mathbf{G}_{invis} - \mathbf{G}_{visc}\right) = 0 \tag{90}$$

$$\implies \frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{F}_{invis}}{\partial \mathbf{x}} + \frac{\partial \mathbf{F}_{visc}}{\partial \mathbf{x}} - \frac{\partial \mathbf{G}_{invis}}{\partial \mathbf{y}} + \frac{\partial \mathbf{G}_{visc}}{\partial \mathbf{y}} \tag{91}$$

Along with the SAT terms for the weak implementation of boundary conditions, the above equation can be written in the vector/CFD form as:

Table 3: Errors in the 2D CNSE code

Nx=Ny	L_2 **error**	$O(h)$
05	1.77325 E -4	—
10	4.39034 E -5	2.01
20	8.13453 E -6	2.43
40	6.92814 E -7	3.55
80	4.72476 E -8	3.87

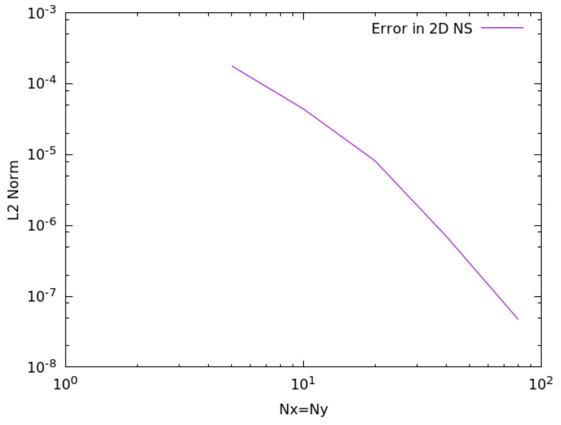


Figure 13: Convergence study of 2D Compressible Navier-Stokes code

$$\frac{\partial}{\partial t}\underbrace{\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho\left(\frac{1}{2}(u^2+v^2)+e\right)\end{bmatrix}}_{\mathbf{u}} = -\frac{\partial}{\partial x}\underbrace{\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ u\rho\left(\frac{1}{2}(u^2+v^2)+e\right)+up\end{bmatrix}}_{\mathbf{F}_{invis}} + \frac{\partial}{\partial x}\underbrace{\begin{bmatrix}0\\ \tau_{xx}\\ \tau_{xy}\\ k\frac{\partial T}{\partial x}+u\tau_{xx}+v\tau_{xy}\end{bmatrix}}_{\mathbf{F}_{visc}} \tag{92}$$

$$-\frac{\partial}{\partial y}\underbrace{\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ v\rho\left(\frac{1}{2}(u^2+v^2)+e\right)+vp\end{bmatrix}}_{\mathbf{G}_{invis}} + \frac{\partial}{\partial y}\underbrace{\begin{bmatrix}0\\ \tau_{yx}\\ \tau_{yy}\\ k\frac{\partial T}{\partial y}+u\tau_{yx}+v\tau_{yy}\end{bmatrix}}_{\mathbf{G}_{visc}} \tag{93}$$

$$-\begin{bmatrix}\mathbf{SAT_{1,left}}\\ \mathbf{SAT_{2,left}}\\ \mathbf{SAT_{3,left}}\\ \mathbf{SAT_{4,left}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,right}}\\ \mathbf{SAT_{2,right}}\\ \mathbf{SAT_{3,right}}\\ \mathbf{SAT_{4,right}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,bottom}}\\ \mathbf{SAT_{2,bottom}}\\ \mathbf{SAT_{3,bottom}}\\ \mathbf{SAT_{3,bottom}}\end{bmatrix} - \begin{bmatrix}\mathbf{SAT_{1,top}}\\ \mathbf{SAT_{2,top}}\\ \mathbf{SAT_{3,top}}\\ \mathbf{SAT_{4,top}}\end{bmatrix} \tag{94}$$

where:

$$e = \frac{p}{(\gamma-1)\rho} \tag{95}$$

$$\tau_{xx} = \lambda\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)+2\mu\frac{\partial u}{\partial x} \tag{96}$$

$$\tau_{yy} = \lambda\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)+2\mu\frac{\partial v}{\partial y} \tag{97}$$

$$\tau_{xy} = \tau_{yx} = \mu\left(\frac{\partial v}{\partial x}+\frac{\partial u}{\partial y}\right) \tag{98}$$

$$\lambda = -\frac{2}{3} \tag{99}$$

From Sutherland's law, dynamic viscosity:

$$\mu = \frac{C_1 \cdot T^{\frac{3}{2}}}{T+S}, \text{ where } C_1 = 1.458\times10^{-6},\ S = 110.4\ K \tag{100}$$

For air at 300 K, usually: $\mu = 1.846\times10^{-5}$ kg/ms

Thermal conductivity:

$$k = \frac{\mu\cdot C_p}{Pr}, \text{where for air at 300 K: } C_p = 1.0049,\ Pr = 0.707 \tag{101}$$

For air at 300 K, usually: $k = 2.6238\times10^{-5}$ kW/mK

For periodic boundary conditions, the SAT terms are exactly same as given in Section 8.
The isentropic vortex case was run with a viscosity of $\mu = 0.00001846$, $\Delta t = 0.0001$, for a total time of $t = 20\Delta t$ to study the error and order of convergence. The results are shown in Table 3 and Figure 13. The solver gave an approximate 3^{rd} order convergence (using 4^{th} order SBP operator with perioidc boundary conditions).

9.1 Problem Description: Lid driven cavity

The benchmark case of a Lid-driven cavity was simulated with the boundary conditions as shown in Figure 14a, for a Reynolds number of 100 and the results are compared with the high-resolution data from Ghia et al. (1982) after steady-state was reached, as shown in Figures 14b and 15.
With $u = 1$,$L = 1$,$\rho = 1$ and $Re = 100$, we get:

$$\mu = \frac{u\rho L}{Re} = \frac{1 \times 1 \times 1}{100} = 0.01 \tag{102}$$

For the SAT boundary conditions, the following target values were chosen for the conserved variables:

$$\underbrace{g_{x_0}}_{For\ SAT_{Left}} = \underbrace{g_{x_n}}_{For\ SAT_{Right}} = \underbrace{g_{y_0}}_{For\ SAT_{Bottom}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2.5 \end{bmatrix} \tag{103}$$

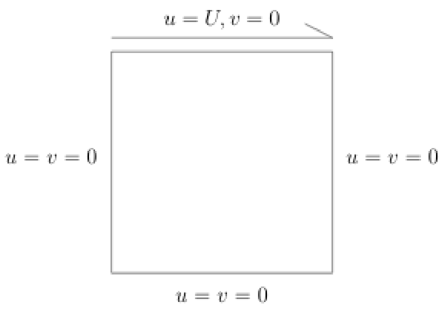
where 2.5 was obtained as:

$$\rho E = \rho \left(\frac{1}{2}\left(u^2 + v^2\right) + \frac{p}{(\gamma - 1)\,\rho} \right) \rightarrow 1 \left(\frac{1}{2}(0+0) + \frac{1}{0.4 \times 1} \right) = 1\,(0 + 2.5) = 2.5 \tag{104}$$

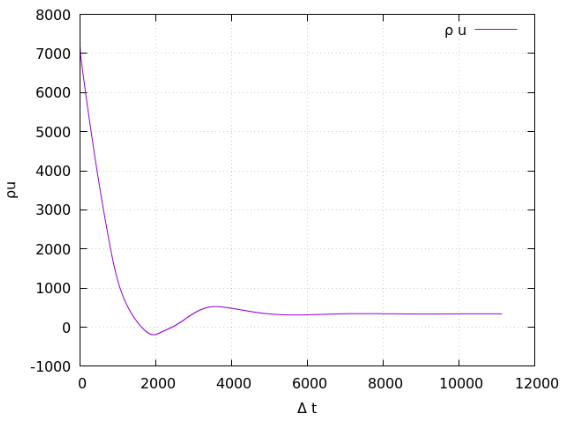
For the top boundary (i.e., driving lid):

$$\underbrace{g_{y_n}}_{SAT_{Top}} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 3 \end{bmatrix} \tag{105}$$

where 3 was obtained by substituting $v = 1$ in Equation (104).



(a) Boundary conditions for the Lid-driven cavity

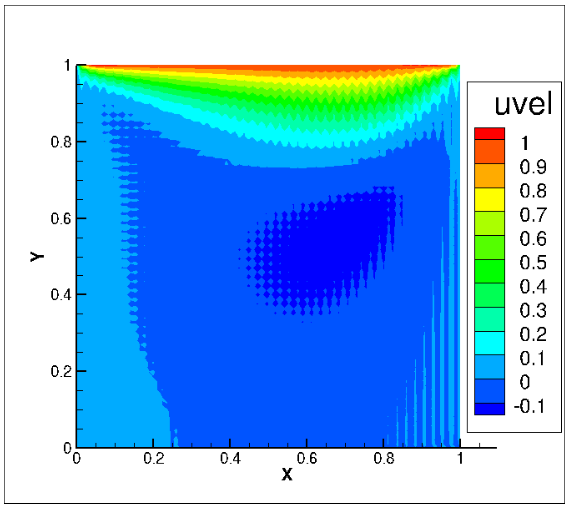


(b) Convergence to steady state by observing $\int_\Omega \rho u$

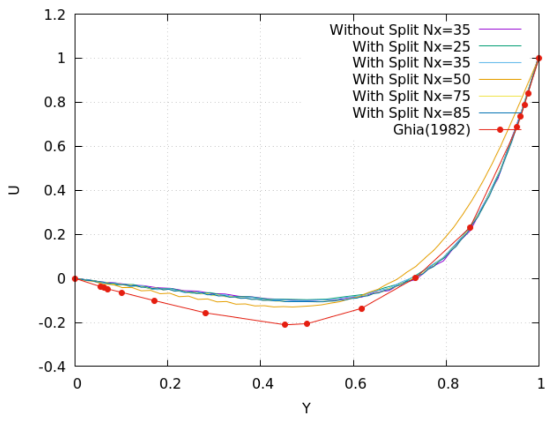
Figure 14: Lid Driven Cavity

9.1.1 Results and Discussion

The solution with the un-split formulation was too unstable that it was unable to reach steady-state for grids finer than $Nx = Ny = 35$, while the simulations run with the split formulation was stable upto about $Nx = Ny = 95$. As shown in the contour in Figure 15a which corresponds to a fine grid of $nx = Ny = 85$, oscillations were still present and are suspected to be caused due to the central scheme that was employed in the SBP method. This might be resolved by using an upwind operator like the one proposed by Mattsson (2017). As seen in Figure 15b, none of the solutions were able to correctly predict the reversed flow that occurred below $y = 0.5$. This might be because just inviscid SATs were used in this study and viscous SATs like the ones suggested by Osusky and Zingg (2013) were not used.



(a) U-velocity contour with Nx=Ny=85



(b) Variation of u-velocity in the y direction at x=0.5

Figure 15: Lid Driven Cavity u-velocity contour

10 Unsuccessful attempts and future works

- Even though an attempt was made to implement artificial dissipation (as seen in Fernández et al. (2014)), it did not work well; probably due to incorrect implementation. Will consider the correct implementation of artifitial dissipation by referring to Mattsson et al. (2007) as suggested in Mattsson (2017). Also consider high-order artificial dissipation operators in the work by Penner and Zingg (2018).
- Even though the upwind SBP derivative matrix proposed in Mattsson (2017) was adopted into the code it did not function properly. Probable errors are: (i) using just Q_+(upwind), and not using both Q_+ and Q_-(downwind) by considering the direction of wind (ii) not adapting SAT implementation for this formulation. Hence, a more careful implementation of the same is planned. More details of SBP-SAT discretization for the isentropic vortex can be found in Svärd et al. (2005).
- Clarify the discrepancy in the splitting of energy equation following the method of Kennedy and Gruber (2008) split fluxes mentioned in Abe et al. (2018) and Gassner et al. (2016). And write a better code following the notation in Abe et al. (2018).
- For the isentropic vortex, since the kinetic energy and entropy jumps each time the vortex crosses the boundary (which eventually grows and diverges the solution), try to implement the entropy stabilization through a diffusion term at the boundaries incorporated in the SAT term as mentioned in Crean et al. (2018) [Eq.22].
- For the lid-driven cavity case, even though the no-slip adiabatic wall boundary condition mentioned in Osusky and Zingg (2013) and Fernández et al. (2014) was applied, the attempt was not successful. In a future work, it is planned to add the adiabatic no-slip SAT on top of the existing invisicd SATs.

11 Summary

In this study, the Summation By Parts (SBP) operator was used to descretize several partial differential equations along with the use of Simultaneous Approximation Term (SAT) boundary conditions starting from the 1-D scalar linear advection all the way upto 2-D Compressible Navier-Stokes equations. The implementation was described with the aid of several matrices and equations. Several hurdles faced during implementation were discussed and their solutions were also provided. The fourth-order central difference SBP operator used here exhibited aliasing errors which destabilized the simulation. An energy-preserving flux-splitting formulation was implemented which helped to suppress the aliasing errors upto a limited extent. The order of accuracy of the codes were checked at several steps and several test problems were simulated including isentropic vortex and lid driven cavity.

12 Acknowledgements

The author gratefully acknowledges the effort taken by Prof.Steven Frankel for introducing the most modern high-order accurate numerical methods through a graduate course in CFD.
The author is thankful to the other members of the CFDLab, namely Vikram Singh, Dr.Viktor Linders, Solal Amouyal, Igal Tsarfis and Dr.Yann Delorme, for providing their valuable insights as well as suggestions for coding, debugging; and also for the several informative and enjoyable discussions.

References

Abe, Y., Morinaka, I., Haga, T., Nonomura, T., Shibata, H., Miyaji, K., 2018. Stable, non-dissipative, and conservative flux-reconstruction schemes in split forms. Journal of Computational Physics 353, 193 – 227. URL: http://www.sciencedirect.com/science/article/pii/S0021999117307453, doi:10.1016/j.jcp.2017.10.007.

Carpenter, M.H., Gottlieb, D., Abarbanel, S., 1994. Time-stable boundary conditions for finite-difference schemes solving hyperbolic systems: Methodology and application to high-order compact schemes. Journal of Computational Physics 111, 220 – 236. URL: https://www.sciencedirect.com/science/article/pii/S0021999184710576, doi:10.1006/jcph.1994.1057.

Crean, J., Hicken, J.E., Fernández, D.C.D.R., Zingg, D.W., Carpenter, M.H., 2017. High-order entropy-stable discretizations of the euler equations for complex geometries. URL: http://oddjob.utias.utoronto.ca/dwz/Miscellaneous/Jared2017Aviation.pdf.

Crean, J., Hicken, J.E., Fernndez, D.C.D.R., Zingg, D.W., Carpenter, M.H., 2018. Entropy-stable summation-by-parts discretization of the euler equations on general curved elements. Journal of Computational Physics 356, 410 – 438. URL: http://www.sciencedirect.com/science/article/pii/S0021999117308999, doi:10.1016/j.jcp.2017.12.015.

Fernández, D.D.R., Hicken, J., Zingg, D., 2014. Review of summation-by-parts operators with simultaneous approximation terms for the numerical solution of partial differential equations. Computers & Fluids 95, 171 – 196. URL: http://www.sciencedirect.com/science/article/pii/S0045793014000796, doi:10.1016/j.compfluid.2014.02.016.

Fisher, T.C., Carpenter, M.H., 2013. High-order entropy stable finite difference schemes for nonlinear conservation laws: Finite domains. Journal of Computational Physics 252, 518 – 557. URL: http://www.sciencedirect.com/science/article/pii/S0021999113004385, doi:10.1016/j.jcp.2013.06.014.

Gassner, G.J., Winters, A.R., Kopriva, D.A., 2016. Split form nodal discontinuous galerkin schemes with summation-by-parts property for the compressible euler equations. Journal of Computational Physics 327, 39 – 66. URL: http://www.sciencedirect.com/science/article/pii/S0021999116304259, doi:10.1016/j.jcp.2016.09.013.

Ghia, U., Ghia, K., Shin, C., 1982. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. Journal of Computational Physics 48, 387 – 411. URL: http://www.sciencedirect.com/science/article/pii/0021999182900584, doi:10.1016/0021-9991(82)90058-4.

Kennedy, C.A., Gruber, A., 2008. Reduced aliasing formulations of the convective terms within the navierstokes equations for a compressible fluid. Journal of Computational Physics 227, 1676 – 1700. URL: http://www.sciencedirect.com/science/article/pii/S0021999107004251, doi:10.1016/j.jcp.2007.09.020.

Lundquist, T., Nordström, J., 2016. Efficient fully discrete summation-by-parts schemes for unsteady flow problems. BIT Numerical Mathematics 56, 951–966. URL: http://www.diva-portal.org/smash/get/diva2:754172/FULLTEXT02, doi:10.1007/s10543-015-0599-0.

Mattsson, K., 2001. Imposing boundary conditions with the injection, the projection and the simultaneous approximation term methods, in: Satofuka, N. (Ed.), Computational Fluid Dynamics 2000, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 343–348. URL: https://link.springer.com/chapter/10.1007/978-3-642-56535-9_50.

Mattsson, K., 2017. Diagonal-norm upwind sbp operators. Journal of Computational Physics 335, 283 – 310. URL: http://www.sciencedirect.com/science/article/pii/S002199911730058X, doi:10.1016/j.jcp.2017.01.042.

Mattsson, K., Svard, M., Carpenter, M., Nordstrm, J., 2007. High-order accurate computations for unsteady aerodynamics. Computers & Fluids 36, 636 – 649. URL: http://www.sciencedirect.com/science/article/pii/S0045793006000636, doi:10.1016/j.compfluid.2006.02.004.

Osusky, M., Zingg, D.W., 2013. Parallel newtonkrylovschur flow solver for the navierstokes equations. AIAA Journal 51(12), 2833 – 2851. URL: https://arc.aiaa.org/doi/10.2514/1.J052487, doi:10.2514/1.J052487.

Penner, D.C., Zingg, D.W., 2018. High-order artificial dissipation operators possessing the summation-by-parts property. URL: http://oddjob.utias.utoronto.ca/dwz/Miscellaneous/Penner2018.pdf, doi:10.2514/6.2018-4165.

Strand, B., 1994. Summation by parts for finite difference approximations for d/dx. Journal of Computational Physics 110, 47 – 67. URL: http://www.sciencedirect.com/science/article/pii/S0021999184710059, doi:10.1006/jcph.1994.1005.

Svärd, M., Mattsson, K., Nordström, J., 2005. Steady-state computations using summation-by-parts operators. Journal of Scientific Computing 24, 79–95. URL: https://doi.org/10.1007/s10915-004-4788-2, doi:10.1007/s10915-004-4788-2.

Yee, H., Vinokur, M., Djomehri, M., 2000. Entropy splitting and numerical dissipation. Journal of Computational Physics 162, 33 – 81. URL: http://www.sciencedirect.com/science/article/pii/S0021999100965173, doi:10.1006/jcph.2000.6517.

13 Computer Code: 2-D Compressible Navier-Stokes Equation in Split-form

```
!**************************************************************************
!**************************************************************************
!!!!**************** SBP - SAT - Split Fluxes ************************!!!!
!**************************************************************************
!**************************************************************************
! This code solves the 2-D Compressible Navier-Stokes equations using
! Summation by Parts (SBP) operators, Simultaneous Approximation Terms (SAT)
! boundary conditions and Split-form fluxes. Second, fourth and sixth-order
! accurate SBP operators are avialable. The default test case selected is the
! Lid Driven cavity at Re=100
! Author: Ronith Stanly
! Lecturer: Prof.Steven Frankel
! Last Updated: 22 February, 2019
! CFD Lab, Technion, Israel
!**************************************************************************
!**************************************************************************


!**************************************************************************
!**************************************************************************
!**************************************************************************
!**************************************************************************
! This module defines the KIND types of all the variables used in the code:
! I4B, I2B and I1B for integer variables, SP and DP for real variables (and
! SPC and DPC for corresponding complex cases), and LGT for the default
! logical type. This follows the convention used the Numerical Recipes for
! Fortran 90 types module 'nrtype', pp. 1361
```

```
MODULE types_vars
  ! Symbolic names for kind types of 4-, 2- and 1-byte integers:
  INTEGER, PARAMETER :: I4B = SELECTED_INT_KIND(9)
  INTEGER, PARAMETER :: I2B = SELECTED_INT_KIND(4)
  INTEGER, PARAMETER :: I1B = SELECTED_INT_KIND(2)
  ! Symbolic names for kind types of single- and double-precison reals
  INTEGER, PARAMETER :: SP = KIND(1.0)
  INTEGER, PARAMETER :: DP = KIND(1.0D0)
  ! Symbolic names for kind types of single- and double-precison complex
  INTEGER, PARAMETER :: SPC = KIND((1.0,1.0))
  INTEGER, PARAMETER :: DPC = KIND((1.0D0,1.0D0))
  ! Symbolic name for kind type of default logical
  INTEGER, PARAMETER :: LOGIC = KIND(.true.)
  ! Frequently used mathematical constants (with precision to spare)
  REAL(DP), PARAMETER :: zero  = 0.0_dp
  REAL(DP), PARAMETER :: half  = 0.5_dp
  REAL(DP), PARAMETER :: one   = 1.0_dp
  REAL(DP), PARAMETER :: two   = 2.0_dp
  REAL(DP), PARAMETER :: three = 3.0_dp
  REAL(DP), PARAMETER :: four  = 4.0_dp
  REAL(DP), PARAMETER :: pi    = 3.141592653589793238462643383279502884197_dp
  REAL(DP), PARAMETER :: pio2  = 1.57079632679489661923132169163975144209858_dp
  REAL(DP), PARAMETER :: twopi = 6.283185307179586476925286766559005768394_dp
END MODULE types_vars

!***************************************************************************
!***************************************************************************
!***************************************************************************
!***************************************************************************
! This module defines and allocates the variables needed in the current simulation
! If needed, add new variables at the beginning of the module, then allocate
! them in the subroutine memalloc
MODULE variables
  USE types_vars
  ! Add new variables here
  INTEGER :: nel, ntimes, nptsx, nptsy, nptst, iwave, tick, nvar, ic_user, porder
  REAL(DP) :: a, b, Dx, Dy, t, cfl, cfl_ip, u_rk3, u_rk4, l2_rk3
  REAL(DP) :: c, d, Dt, gam, rgas
  REAL(DP) :: ay, by, alpha_split, beta_split

  !1D arrays
  REAL(DP), ALLOCATABLE, DIMENSION(:) :: x, y, time, h, l2_rk4
  REAL(DP), ALLOCATABLE, DIMENSION(:) :: e_x_0, e_x_n, e_y_0, e_y_n

  !2D arrays
  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: Hmatinv, Qmat, Mmat, Smat, Bmat, D1mat, D2mat, Dx1, Q_tran, Hinvx, Hinvy
  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: Dy1, g_x_0, g_x_n, ke, temp, sos, mach, rho, uvel, vvel, pre, e, et
  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: g_y_0, g_y_n, correction

  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: rho_u, f2, df2dx, f6, df6dx, f7, df7dx
  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: rho_v, g2, dg2dy, g6, dg6dy, g7, dg7dy

  !3D arrays
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: f, u, uold, dfdx, k1, k2, k3, k4, u_ic, u_dd, rhs, u_e, dfdx_e
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: g, dgdy, temp1d, SAT_x0, SAT_xn, dgdy_e, SAT_y0, SAT_yn
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: f_visc, g_visc, dfdx_visc, dgdy_visc

  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: phi, f1, df1dx, split_x1, split_x2, split_x3, split_x4
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: split_x5, split_x6, split_x7, f3, df3dx, df4dx, f5, df5dx
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: pre_mat_x, dpdx,f4
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: g1, dg1dy, split_y1, split_y2, split_y3, split_y4
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: split_y5, split_y6, split_y7, g3, dg3dy, dg4dy, g5, dg5dy
  REAL(DP), ALLOCATABLE, DIMENSION(:,:,:) :: pre_mat_y, dpdy, g4
  CONTAINS


!************* Allocate memory******************
  SUBROUTINE memalloc
    ! Allocate memory for grid, solution, and flux function

    !1D arrays
    ALLOCATE(h(1:2),l2_rk4(1:2))
    ALLOCATE(x(0:nptsx),e_x_0(0:nptsx),e_x_n(0:nptsx),e_y_0(0:nptsy),e_y_n(0:nptsy))

    ALLOCATE(y(0:nptsy))
```

```
      !2D arrays
      ALLOCATE(Dx1(0:nptsx,0:nptsx),Hinvx(0:nptsx,0:nptsx),Hinvy(0:nptsy,0:nptsy))

      ALLOCATE(rho(0:nptsx,0:nptsy),uvel(0:nptsx,0:nptsy),vvel(0:nptsx,0:nptsy),pre(0:nptsx,0:nptsy))
      ALLOCATE(e(0:nptsx,0:nptsy),et(0:nptsx,0:nptsy),Dy1(0:nptsy,0:nptsy))
      ALLOCATE(g_x_0(1:nvar,0:nptsy),g_x_n(1:nvar,0:nptsy))
      ALLOCATE(ke(0:nptsx,0:nptsy),temp(0:nptsx,0:nptsy),sos(0:nptsx,0:nptsy),mach(0:nptsx,0:nptsy))
      ALLOCATE(g_y_0(1:nvar,0:nptsx),g_y_n(1:nvar,0:nptsx))

      ALLOCATE(rho_u(0:nptsx,0:nptsy),f2(0:nptsx,0:nptsy))
      ALLOCATE(df2dx(0:nptsx,0:nptsy),f6(0:nptsx,0:nptsy),df6dx(0:nptsx,0:nptsy),f7(0:nptsx,0:nptsy))
      ALLOCATE(df7dx(0:nptsx,0:nptsy))
      ALLOCATE(rho_v(0:nptsx,0:nptsy),g2(0:nptsx,0:nptsy),dg2dy(0:nptsx,0:nptsy),g6(0:nptsx,0:nptsy))
      ALLOCATE(dg6dy(0:nptsx,0:nptsy),g7(0:nptsx,0:nptsy),dg7dy(0:nptsx,0:nptsy))

      !3D arrays
      ALLOCATE(f(1:nvar,0:nptsx,0:nptsy),u(1:nvar,0:nptsx,0:nptsy),u_ic(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(u_dd(1:nvar,0:nptsx,0:nptsy),uold(1:nvar,0:nptsx,0:nptsy),dfdx(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(k1(1:nvar,0:nptsx,0:nptsy),k2(1:nvar,0:nptsx,0:nptsy),k3(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(k4(1:nvar,0:nptsx,0:nptsy),g(1:nvar,0:nptsx,0:nptsy),rhs(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(u_e(1:nvar,0:nptsx,0:nptsy),dfdx_e(1:nvar,0:nptsx,0:nptsy),dgdy(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(SAT_x0(1:nvar,0:nptsx,0:nptsy),SAT_xn(1:nvar,0:nptsx,0:nptsy),dgdy_e(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(SAT_y0(1:nvar,0:nptsx,0:nptsy),SAT_yn(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(f_visc(1:nvar,0:nptsx,0:nptsy),g_visc(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(dfdx_visc(1:nvar,0:nptsx,0:nptsy),dgdy_visc(1:nvar,0:nptsx,0:nptsy))

      ALLOCATE(phi(1:nvar,0:nptsx,0:nptsy),f1(1:nvar,0:nptsx,0:nptsy),df1dx(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(split_x1(1:nvar,0:nptsx,0:nptsy),split_x2(1:nvar,0:nptsx,0:nptsy),split_x3(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(split_x4(1:nvar,0:nptsx,0:nptsy),split_x5(1:nvar,0:nptsx,0:nptsy),split_x6(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(split_x7(1:nvar,0:nptsx,0:nptsy),f3(1:nvar,0:nptsx,0:nptsy),df3dx(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(f4(1:nvar,0:nptsx,0:nptsy),df4dx(1:nvar,0:nptsx,0:nptsy),f5(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(df5dx(1:nvar,0:nptsx,0:nptsy),pre_mat_x(1:nvar,0:nptsx,0:nptsy),dpdx(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(g1(1:nvar,0:nptsx,0:nptsy),dg1dy(1:nvar,0:nptsx,0:nptsy),split_y1(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(split_y2(1:nvar,0:nptsx,0:nptsy),split_y3(1:nvar,0:nptsx,0:nptsy),split_y4(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(split_y5(1:nvar,0:nptsx,0:nptsy),split_y6(1:nvar,0:nptsx,0:nptsy),split_y7(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(g3(1:nvar,0:nptsx,0:nptsy),dg3dy(1:nvar,0:nptsx,0:nptsy),g4(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(dg4dy(1:nvar,0:nptsx,0:nptsy),g5(1:nvar,0:nptsx,0:nptsy),dg5dy(1:nvar,0:nptsx,0:nptsy))
      ALLOCATE(pre_mat_y(1:nvar,0:nptsx,0:nptsy),dpdy(1:nvar,0:nptsx,0:nptsy))

    END SUBROUTINE memalloc


!************* Deallocate memory (end of the program)*******************
    SUBROUTINE dealloc
      ! Deallocate memory for grid, solution, and flux function
      DEALLOCATE(x,u,f,uold,dfdx,k1,k2,k3,k4,u_dd,u_e)
    END SUBROUTINE dealloc

  END MODULE variables

  !*************************************************************************
  !*************************************************************************
  MODULE subroutines
    USE types_vars
    USE variables
    CONTAINS

!************* Get the inputs *******************
   SUBROUTINE inputs
    IMPLICIT NONE  ! Forces explicit type declaration to avoid errors

    ! Assume 2D domain is x[-5,5],y[-5,5] and time
    a = 0.0d0; b = 1.0d0; ay=0.0d0; by=1.0d0; c = 0.0d0; d = 6.0d0
    ! Kennedy & Gruber Split parameters
    alpha_split=0.25d0; beta_split=0.25d0

    ! Number of variables (or no. of eqs. in the system), here 2 (for 2D Navier-Stokes it is 4)
    nvar=4
    gam=1.4d0 ! gamma
  !  rgas=8.314d0 ! Gas constant from ideal gas law
    rgas=8.314598/28.97d0

    ! Read from screen input information for program
    WRITE(*,*) 'Assuming equal number of cewlls in both directions'
    WRITE(*,*) 'Please input the number of cells/volumes/elements:'
    READ(*,*) nel
```

```fortran
    nptsx = nel
    nptsy = nel

    WRITE(*,*) 'Please input the desired CFL number'
    READ(*,*) cfl
    cfl_ip=cfl
    Dx = (b-a)/FLOAT(nptsx)
    Dy= (by-ay)/FLOAT(nptsy)
    !Dt = (cfl_ip*Dx)
    !Dt = (cfl_ip * Dx)/a
    !nptst = ABS(d-c)/Dt

    WRITE(*,*) 'Enter preferred order of accuracy 2,4 or 6?'
    READ(*,*) porder

    WRITE(*,*) 'Time-step size=', Dt
    WRITE(*,*) 'Domain will be discretized with ', nptsx**2
    WRITE(*,*) 'for the given CFL number=', cfl_ip

    ! Echo print your input to make sure it is correct
    WRITE(*,*) 'Your 2D domain is from, x=', a, ' to ', b, 'y=',ay,' to ',by
    WRITE(*,*) 'and time is from ',c,'s to ', d,'s'

  END SUBROUTINE inputs

!*********** Generate a 2D grid ***************
    SUBROUTINE grid2d
    IMPLICIT NONE
    INTEGER :: i,j

    ! Generate grid
    h(1)=Dx
    h(2)=Dy
    DO i = 0, nptsx
      x(i) = a + i*Dx
    END DO

    DO j=0, nptsy
      y(j) = ay + j*Dy
    END DO

    END SUBROUTINE grid2d


!************** Provide intial condition ************************
!***************** Lid-Driven Cavity ****************************
    SUBROUTINE init2d
      IMPLICIT NONE
      INTEGER :: i, j
      REAL, DIMENSION(:,:) :: rad(0:nptsx,0:nptsy)

      DO i=0, nptsx
        DO j=0, nptsy
          rho(i,j)=1.0d0

          uvel(i,j)=1.0d0
          vvel(i,j)=0.0d0

          pre(i,j)=1.0d0

          sos=SQRT(gam*pre(i,j)/rho(i,j))

        END DO
      END DO

    Dt=cfl_ip*MIN(Dx/MAXVAL(uvel+sos),Dy/MAXVAL(vvel+sos))
    nptst = ABS(d-c)/Dt

    OPEN(1, file = 'initial.dat', status = 'replace')
    WRITE(1,'(A)') 'VARIABLES = "X","Y","rho","uvel","vvel","pre"'
    WRITE(1,*) 'ZONE I=',nptsx+1,', J=',nptsy+1,', ZONETYPE=ORDERED,'
    WRITE(1,*) 'DATAPACKING=POINT, SOLUTIONTIME=0.0'
    DO i = 0, nptsx
      DO j=0, nptsy
        WRITE(1, *) x(i), y(j), rho(i,j), uvel(i,j), vvel(i,j), pre(i,j)
      END DO
    END DO
    CLOSE(1)

    END SUBROUTINE init2d


!************** Construct solution vector ************************
    SUBROUTINE solvec
      IMPLICIT NONE
      INTEGER :: i,j

      DO i = 0, nptsx
        DO j=0, nptsy
          e(i,j)=pre(i,j)/((gam-1.0d0))
          et(i,j)=0.5d0*(uvel(i,j)**2.0d0+vvel(i,j)**2.0d0) + e(i,j)
          u(1,i,j)=rho(i,j)
          u(2,i,j)=rho(i,j)*uvel(i,j)
          u(3,i,j)=rho(i,j)*vvel(i,j)
          u(4,i,j)=rho(i,j)*et(i,j)
        END DO
      END DO

      OPEN(20, file = 'initial_solvec.dat', status = 'replace')
      WRITE(20,'(A)') 'VARIABLES = "X","Y","u1","u2","u3"'
      WRITE(20,*) 'ZONE I=',nptsx+1,', J=',nptsy+1,', ZONETYPE=ORDERED,'
      WRITE(20,*) 'DATAPACKING=POINT, SOLUTIONTIME=0.0'
      DO i = 0, nptsx
        DO j=0, nptsy
          WRITE(20, *) x(i), y(j), u(1,i,j), u(2,i,j), u(3,i,j)
        END DO
      END DO
      CLOSE(20)

      u_ic=u ! To compute l2 with initial condition

    END SUBROUTINE solvec

!************** Compute viscous flux vectors f_visc and g_visc ************************
    SUBROUTINE flux_visc
      IMPLICIT NONE
      INTEGER :: i, j
      REAL :: C1, S, Cp, Pr
      ! 1-D arrays
      REAL, DIMENSION(:) :: termx(0:nptsx), termx1(0:nptsx)
      REAL, DIMENSION(:) :: termy(0:nptsy), termy1(0:nptsy)
      REAL, DIMENSION(:) :: mu(0:nptsx,0:nptsy),kappa(0:nptsx,0:nptsy),lambda(0:nptsx,0:nptsy)
      ! 2-D arrays
      REAL, DIMENSION(:,:) :: tau_xx(0:nptsx,0:nptsy),tau_xy(0:nptsx,0:nptsy)
      REAL, DIMENSION(:,:) :: tau_yx(0:nptsx,0:nptsy),tau_yy(0:nptsx,0:nptsy)
      REAL, DIMENSION(:,:) :: dudx(0:nptsx,0:nptsy),dvdy(0:nptsx,0:nptsy)
      REAL, DIMENSION(:,:) :: dvdx(0:nptsx,0:nptsy),dudy(0:nptsx,0:nptsy)
      REAL, DIMENSION(:,:) :: dtdx(0:nptsx,0:nptsy),dtdy(0:nptsx,0:nptsy)

      ! Coefficients to compute dynamic viscousity
      C1=1.458d0*10.d0**(-6.0d0)
      S=110.4d0
      Cp=1.0049d0 ! For air at 300K
      Pr=0.707d0  ! For air at 300K

      DO i=0, nptsx
        DO j=0, nptsy
          ! Molecular viscosity
           mu(i,j)=0.01d0 ! Re=100
          ! Bulk viscosity
          lambda(i,j)=-(2.0d0/3.0d0)*mu(i,j)
          ! Thermal conductivity
          kappa(i,j)=(mu(i,j)*Cp)/Pr
        END DO
      END DO

      ! dudx
      DO j=0, nptsy
        termx(0:nptsx)=uvel(0:nptsx,j)
        termx1(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx(0:nptsx))
        dudx(0:nptsx,j)=termx1(0:nptsx)
      END DO
      ! dvdx
      DO j=0, nptsy
        termx(0:nptsx)=vvel(0:nptsx,j)
        termx1(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx(0:nptsx))
        dvdx(0:nptsx,j)=termx1(0:nptsx)
      END DO

      ! dvdy
      DO i=0, nptsx
        termy(0:nptsy)=vvel(i,0:nptsy)
        termy1(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy(0:nptsy))
        dvdy(i,0:nptsy)=termy1(0:nptsy)
      END DO

      ! dudy
      DO i=0, nptsx
        termy(0:nptsy)=uvel(i,0:nptsy)
        termy1(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy(0:nptsy))
        dudy(i,0:nptsy)=termy1(0:nptsy)
      END DO

      ! Stresses
      DO i=0, nptsx
        DO j=0, nptsy
          ! Normal Stresses
          tau_xx(i,j)=lambda(i,j)*(dudx(i,j)+dvdy(i,j))+2.0d0*mu(i,j)*dudx(i,j)
          tau_yy(i,j)=lambda(i,j)*(dudx(i,j)+dvdy(i,j))+2.0d0*mu(i,j)*dvdy(i,j)
          ! Shear stresses
          tau_xy(i,j)=mu(i,j)*(dvdx(i,j)+dudy(i,j))
          tau_yx(i,j)=tau_xy(i,j)
        END DO
      END DO

      ! Thermal gradients
      ! dtdx
      DO j=0, nptsy
        termx(0:nptsx)=temp(0:nptsx,j)
        termx1(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx(0:nptsx))
        dtdx(0:nptsx,j)=termx1(0:nptsx)
      END DO

      ! dtdy
      DO i=0, nptsx
        termy(0:nptsy)=temp(i,0:nptsy)
        termy1(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy(0:nptsy))
        dtdy(i,0:nptsy)=termy1(0:nptsy)
      END DO

      ! viscous fluxes
      DO i=0, nptsx
        DO j=0, nptsy
        f_visc(1,i,j)=0.0d0
        f_visc(2,i,j)=tau_xx(i,j)
        f_visc(3,i,j)=tau_xy(i,j)
        f_visc(4,i,j)=kappa(i,j)*dtdx(i,j)+uvel(i,j)*tau_xx(i,j)+vvel(i,j)*tau_xy(i,j)

        g_visc(1,i,j)=0.0d0
        g_visc(2,i,j)=tau_yx(i,j)
        g_visc(3,i,j)=tau_yy(i,j)
        g_visc(4,i,j)=kappa(i,j)*dtdy(i,j)+uvel(i,j)*tau_yx(i,j)+vvel(i,j)*tau_yy(i,j)
        END DO
      END DO

    END SUBROUTINE flux_visc


!-------------------------------------------------------

  SUBROUTINE matrix(idir,npts)
  IMPLICIT NONE
  INTEGER :: i,j
  INTEGER :: idir, npts
  REAL(DP) :: x1

  allocate (Hmatinv(0:npts,0:npts),Qmat(0:npts,0:npts))
  allocate (Mmat(0:npts,0:npts),Smat(0:npts,0:npts))
  allocate (Bmat(0:npts,0:npts))
  allocate (D1mat(0:npts,0:npts),D2mat(0:npts,0:npts))

! Construct scheme matrices

  select case (porder)
!****** 2nd order accurate *****
  case(2)
! Hmatrixinv
  Hmatinv = 0.0
  Hmatinv(0,0) = 2.0/h(idir)
  do i = 1,npts-1
    Hmatinv(i,i) = 1.0/h(idir)
  end do
  Hmatinv(npts,npts) = 2.0/h(idir)

! Qmatrix
  Qmat = 0.0
  Qmat(0,0) = -0.5
  Qmat(0,1) = 0.5
  do i = 1, npts-1
    Qmat(i,i-1) = -0.5
    Qmat(i,i) = 0.0
    Qmat(i,i+1) = 0.5
  end do
  Qmat(npts,npts-1) = -0.5
  Qmat(npts,npts) = 0.5

! Mmatrix
  Mmat = 0.0
  Mmat(0,0) = 1.0
  Mmat(0,1) = -1.0
  Mmat(1,0) = -1.0
  do i = 1, npts-1
    Mmat(i,i-1) = -1.0
    Mmat(i,i) = 2.0
    Mmat(i,i+1) = -1.0
  end do
  Mmat(npts,npts-1) = -1.0
  Mmat(npts,npts) = 1.0
  Mmat(:,:) = Mmat(:,:)/h(idir)

! Bmatrix
  Bmat = 0.0
  Bmat(0,0) = -1.0
  Bmat(npts,npts) = 1.0

! Smatrix
  Smat = 0.0
  Smat(0,0) = 3./2.
  Smat(0,1) = -2.0
  Smat(0,2) = 0.5
  do i = 1,npts-1
    Smat(i,i) = 1.0
  end do
  Smat(npts,npts) = 3./2.
  Smat(npts,npts-1) = -2.0
  Smat(npts,npts-2) = 0.5
  Smat(:,:) = Smat(:,:)/h(idir)

! D1matrix
  D1mat = matmul(Hmatinv,Qmat)
! D2matrix
  D2mat = matmul(Hmatinv,(-Mmat+matmul(Bmat,Smat)))


!****** 4th order accurate *****
  case(4)
! Hmatinv
  Hmatinv = 0.0
  Hmatinv(0,0) = 1./(17./48.)
  Hmatinv(1,1) = 1./(59./48.)
  Hmatinv(2,2) = 1./(43./48.)
  Hmatinv(3,3) = 1./(49./48.)
  do i = 4, npts-4
```

```
    Hmatinv(i,i) = 1.0
  end do
  do i = 0, 3
    Hmatinv(npts-i,npts-i) = Hmatinv(i,i)
  end do
  Hmatinv = Hmatinv/h(idir)

! Qmatrix
  Qmat = 0.0
  Qmat(0,0) = -0.5
  Qmat(0,1) = 59./96.
  Qmat(0,2) = -1./12.
  Qmat(0,3) = -1./32.
  Qmat(1,0) = -59./96.
  Qmat(2,0) = 1./12.
  Qmat(3,0) = 1./32.
  Qmat(1,1) = 0.0
  Qmat(1,2) = 59./96.
  Qmat(2,1) = -59./96.
  Qmat(2,2) = 0.0
  Qmat(2,3) = 59./96.
  Qmat(2,4) = -1./12.
  Qmat(3,1) = 0.0
  Qmat(3,2) = -59./96.
  Qmat(3,3) = 0.0
  Qmat(3,4) = 2./3.
  Qmat(3,5) = -1./12.
  do i = 4, npts-4
    Qmat(i,i-2) = 1./12.
    Qmat(i,i-1) = -2./3.
    Qmat(i,i) = 0.0
    Qmat(i,i+1) = 2./3.
    Qmat(i,i+2) = -1./12.
  end do
  Qmat(npts,npts) = 0.5
  Qmat(npts,npts-1) = -59./96.
  Qmat(npts,npts-2) = 1./12.
  Qmat(npts,npts-3) = 1./32.
  Qmat(npts-1,npts) = 59./96.
  Qmat(npts-1,npts-2) = -59./96.
  Qmat(npts-2,npts) = -1./12.
  Qmat(npts-2,npts-1) = 59./96.
  Qmat(npts-2,npts-3) = -59./96.
  Qmat(npts-2,npts-4) = 1./12.
  Qmat(npts-3,npts) = -1./32.
  Qmat(npts-3,npts-2) = 59./96.
  Qmat(npts-3,npts-4) = -2./3.
  Qmat(npts-3,npts-5) = 1./12.

! D1matrix
  D1mat = matmul(Hmatinv,Qmat)

! Dmat2
  D2mat = 0.0
  D2mat(0,0) = 2.0
  D2mat(0,1) = -5.0
  D2mat(0,2) = 4.0
  D2mat(0,3) = -1.0
  D2mat(1,0) = 1.0
  D2mat(2,0) = -4./43.
  D2mat(3,0) = -1./49.
  D2mat(1,1) = -2.0
  D2mat(1,2) = 1.0
  D2mat(1,3) = 0.0
  D2mat(2,1) = 59./43.
  D2mat(2,2) = -110./43.
  D2mat(2,3) = 59./43.
  D2mat(2,4) = -4./43.
  D2mat(3,1) = 0.0
  D2mat(3,2) = 59./49.
  D2mat(3,3) = -118./49.
  D2mat(3,4) = 64./49.
  D2mat(3,5) = -4./49.
  do i = 4,npts
    D2mat(i,i) = 4./3.
    D2mat(i,i-1) = -1./12.
  end do
  do i = 4,npts-3
    D2mat(i,i+1) = -5./2.
    D2mat(i,i+2) = 4./3.
    D2mat(i,i+3) = 1./12.
  end do
  D2mat(npts-1,npts) = -5./2.
  D2mat(npts-2,npts) = 4./3.
  D2mat(npts-2,npts-1) = -5./2.

  D2mat = D2mat/h(idir)**2

! Smat
  Smat = 0.0
  Smat(0,0) = 11./6.
  Smat(0,1) = -3.
  Smat(0,2) = 3./2.
  Smat(0,3) = -1./3.
  Smat(1,1) = 1.0
  Smat(npts,npts) = 11./6.
  Smat(npts,npts-1) = -3.
  Smat(npts,npts-2) = 3./2.
  Smat(npts,npts-3) = -1./3.
  Smat(npts-1,npts-1) = 1.0
  Smat = Smat/h(idir)

!****** 6th order accurate *****
  case(6)
! Hmatinv
  Hmatinv = 0.0
  Hmatinv(0,0) = 1./(13649./43200.)
  Hmatinv(1,1) = 1./(12013./8640.)
  Hmatinv(2,2) = 1./(2711./4320.)
  Hmatinv(3,3) = 1./(5359./4320.)
  Hmatinv(4,4) = 1./(7877./8640.)
  Hmatinv(5,5) = 1./(43801./43200.)
  do i = 6, npts-6
    Hmatinv(i,i) = 1.0
  end do
  do i = 0, 5
    Hmatinv(npts-i,npts-i) = Hmatinv(i,i)
  end do
  Hmatinv = Hmatinv/h(idir)

! Qmatrix
  x1 = 342523./518400.
  Qmat = 0.0
  Qmat(0,0) = -0.5
  Qmat(0,1) = x1 - 953./16200.
  Qmat(1,0) = -Qmat(0,1)
  Qmat(0,2) = -4.0*x1 + 715489./259200.
  Qmat(2,0) = -Qmat(0,2)
  Qmat(0,3) = 6.0*x1 - 62639./14400.
  Qmat(3,0) = -Qmat(0,3)
  Qmat(0,4) = -4.0*x1 + 147127./51840.
  Qmat(4,0) = -Qmat(0,4)
  Qmat(0,5) = x1 - 89387./129600.
  Qmat(5,0) = -Qmat(0,5)
  Qmat(1,2) = 10.*x1 - 57139./8640.
  Qmat(2,1) = -Qmat(1,2)
  Qmat(1,3) = -20.*x1 + 745733./51840.
  Qmat(3,1) = -Qmat(1,3)
  Qmat(1,4) = 15.*x1 - 18343./1728.
  Qmat(4,1) = -Qmat(1,4)
  Qmat(1,5) = -4.*x1 + 240569./86400.
  Qmat(5,1) = -Qmat(1,5)
  Qmat(2,3) = 20.*x1 - 176839./12960.
  Qmat(3,2) = -Qmat(2,3)
  Qmat(2,4) = -20.*x1 + 242111./17280.
  Qmat(4,2) = -Qmat(2,4)
  Qmat(2,5) = 6.*x1 - 182261./43200.
  Qmat(5,2) = -Qmat(2,5)
  Qmat(3,4) = 10.*x1 - 165041./25920.
  Qmat(4,3) = -Qmat(3,4)
  Qmat(3,5) = -4.*x1 + 710473./259200.
  Qmat(5,3) = -Qmat(3,5)
  Qmat(3,6) = 1./60.
  Qmat(6,3) = -Qmat(3,6)
  Qmat(4,5) = x1
  Qmat(5,4) = -Qmat(4,5)
  Qmat(4,6) = -3./20.
  Qmat(4,7) = 1./60.
  Qmat(5,6) = 3./4.
  Qmat(5,7) = -3./20.
  Qmat(5,8) = 1./60.

  do i = 6, npts-6
    Qmat(i,i-3) = -1./60.
    Qmat(i,i-2) = 3./20.
    Qmat(i,i-1) = -3./4.
    Qmat(i,i) = 0.0
    Qmat(i,i+1) = 3./4.
    Qmat(i,i+2) = -3./20.
    Qmat(i,i+3) = 1./60.
  end do
  do i = 0,5
    do j = 0,5
      Qmat(npts-i,npts-j) = -Qmat(i,j)
    end do
  end do
  Qmat(npts-3,npts-6) = -Qmat(3,6)
  Qmat(npts-4,npts-6) = -Qmat(4,6)
  Qmat(npts-4,npts-7) = -Qmat(4,7)
  Qmat(npts-5,npts-6) = -Qmat(5,6)
  Qmat(npts-5,npts-7) = -Qmat(5,7)
  Qmat(npts-5,npts-8) = -Qmat(5,8)

! D1matrix
  D1mat = matmul(Hmatinv,Qmat)

  end select

  END SUBROUTINE matrix


!************** Compute flux vector in x *************************
    SUBROUTINE flux
    IMPLICIT NONE
      INTEGER :: i, j, ivar
      REAL, DIMENSION(:) :: termx(0:nptsx), termx1(0:nptsx), termx2(0:nptsx), termx3(0:nptsx)
      REAL, DIMENSION(:) :: termx4(0:nptsx), termx5(0:nptsx), termx6(0:nptsx), termx7(0:nptsx)
      REAL, DIMENSION(:) :: termx8(0:nptsx), termx9(0:nptsx), termx10(0:nptsx), termx11(0:nptsx)
      REAL, DIMENSION(:) :: termx12(0:nptsx), termx13(0:nptsx)

       phi(1,0:nptsx,0:nptsy)=1.0d0
       phi(2,0:nptsx,0:nptsy)=uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy) ! u
       phi(3,0:nptsx,0:nptsy)=uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy) ! v
       phi(4,0:nptsx,0:nptsy)=0.5d0*((uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy))**2.0d0+ &
                                     (uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy))**2.0d0)

       rho_u(0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)

!********** Compute 1st split flux term *********
       DO ivar=1, nvar
         f1(ivar,0:nptsx,0:nptsy)=rho_u(0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy)
       END DO

       DO ivar=1, nvar
         DO j=0, nptsy
          termx(0:nptsx)=f1(ivar,0:nptsx,j)
          termx1(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx(0:nptsx))
          df1dx(ivar,0:nptsx,j)=termx1(0:nptsx)
         END DO
       END DO

       split_x1(1:nvar,0:nptsx,0:nptsy)=df1dx(1:nvar,0:nptsx,0:nptsy)

!********** Compute 2nd split flux term *********
       f2(0:nptsx,0:nptsy)=rho_u(0:nptsx,0:nptsy)

       DO j=0, nptsy
         termx2(0:nptsx)=f2(0:nptsx,j)
         termx3(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx2(0:nptsx))
         df2dx(0:nptsx,j)=termx3(0:nptsx)
       END DO

       DO ivar=1, nvar
         split_x2(ivar,0:nptsx,0:nptsy)=phi(ivar,0:nptsx,0:nptsy)*df2dx(0:nptsx,0:nptsy)
       END DO

!********** Compute 3rd split flux term *********
       DO ivar=1,nvar
         f3(ivar,0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy) ! rho*phi
       END DO

       DO ivar=1, nvar
         DO j=0, nptsy
          termx4(0:nptsx)=f3(ivar,0:nptsx,j)
          termx5(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx4(0:nptsx))
          df3dx(ivar,0:nptsx,j)=termx5(0:nptsx)
         END DO
       END DO

       DO ivar=1, nvar
         split_x3(ivar,0:nptsx,0:nptsy)=uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*df3dx(ivar,0:nptsx,0:nptsy) ! u*d
       END DO

!********** Compute 4th split flux term *********
       DO ivar=1,nvar
         f4(ivar,0:nptsx,0:nptsy)=uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy) ! u*phi
       END DO

       DO ivar=1, nvar
         DO j=0, nptsy
          termx6(0:nptsx)=f4(ivar,0:nptsx,j)
          termx7(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx6(0:nptsx))
          df4dx(ivar,0:nptsx,j)=termx7(0:nptsx)
         END DO
       END DO

       DO ivar=1, nvar
         split_x4(ivar,0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*df4dx(ivar,0:nptsx,0:nptsy) ! rho*df4dx
       END DO

!********** Compute 5th split flux term *********
       DO ivar=1,nvar
         f5(ivar,0:nptsx,0:nptsy)=phi(ivar,0:nptsx,0:nptsy) ! phi
       END DO

       DO ivar=1, nvar
         DO j=0, nptsy
          termx8(0:nptsx)=f5(ivar,0:nptsx,j)
          termx9(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx8(0:nptsx))
          df5dx(ivar,0:nptsx,j)=termx9(0:nptsx)
         END DO
       END DO

       DO ivar=1, nvar
         split_x5(ivar,0:nptsx,0:nptsy)=rho_u(0:nptsx,0:nptsy)*df5dx(ivar,0:nptsx,0:nptsy) ! rho_u*df5dx
       END DO

!********** Compute 6th split flux term *********
       f6(0:nptsx,0:nptsy)=uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy) ! u

       DO j=0, nptsy
          termx10(0:nptsx)=f6(0:nptsx,j)
          termx11(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx10(0:nptsx))
          df6dx(0:nptsx,j)=termx11(0:nptsx)
       END DO

       DO ivar=1, nvar
         split_x6(ivar,0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy)*df6dx(0:nptsx,0:nptsy) ! rho*ph
       END DO

!********** Compute 7th split flux term *********
        f7(0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy) ! rho

       DO j=0, nptsy
```

```
          termx12(0:nptsx)=f7(0:nptsx,j)
          termx13(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx12(0:nptsx))
          df7dx(0:nptsx,j)=termx13(0:nptsx)
       END DO

       DO ivar=1, nvar
         split_x7(ivar,0:nptsx,0:nptsy)=uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy) &
                                                                                         *df7dx(0:nptsx,0:nptsy) ! u*phi*df7dx
       END DO

       dfdx= alpha_split*split_x1 + beta_split*(split_x2+split_x3+split_x4) + &
             (1-alpha_split-2*beta_split)*(split_x5+split_x6+split_x7)

    END SUBROUTINE flux


!************** Derivative of pressure terms with x *************
     SUBROUTINE dpre_dx
      INTEGER :: i, j, ivar, k
      REAL, DIMENSION(:) :: termx14(0:nptsx), termx15(0:nptsx)

     pre_mat_x(1,0:nptsx,0:nptsy)=0.0d0
     pre_mat_x(2,0:nptsx,0:nptsy)=pre(0:nptsx,0:nptsy)
     pre_mat_x(3,0:nptsx,0:nptsy)=0.0d0
     pre_mat_x(4,0:nptsx,0:nptsy)=(uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*pre(0:nptsx,0:nptsy))*(1.0d0/(gam-1.0d(

      DO ivar=1, nvar
       DO j=0, nptsy
          termx14(0:nptsx)=pre_mat_x(ivar,0:nptsx,j)
          termx15(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx14(0:nptsx))
          dpdx(ivar,0:nptsx,j)=termx15(0:nptsx)
       END DO
      END DO

    END SUBROUTINE dpre_dx


!************** Compute flux vector in y *************************
    SUBROUTINE glux
    IMPLICIT NONE
      INTEGER :: i, j, ivar
      REAL, DIMENSION(:) :: termy(0:nptsy), termy1(0:nptsy), termy2(0:nptsy), termy3(0:nptsy)
      REAL, DIMENSION(:) :: termy4(0:nptsy), termy5(0:nptsy), termy6(0:nptsy), termy7(0:nptsy)
      REAL, DIMENSION(:) :: termy8(0:nptsy), termy9(0:nptsy), termy10(0:nptsy), termy11(0:nptsy)
      REAL, DIMENSION(:) :: termy12(0:nptsy), termy13(0:nptsy)

       phi(1,0:nptsx,0:nptsy)=1.0d0
       phi(2,0:nptsx,0:nptsy)=uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)  ! u
       phi(3,0:nptsx,0:nptsy)=uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)  ! v
       phi(4,0:nptsx,0:nptsy)=0.5d0*((uold(2,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy))**2.0d0+ &
                                     (uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy))**2.0d0)

       rho_v(0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)

!********** Compute 1st split flux term *********
       DO ivar=1, nvar
         g1(ivar,0:nptsx,0:nptsy)=rho_v(0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy)
       END DO

       DO ivar=1, nvar
         DO i=0, nptsx
          termy(0:nptsy)=g1(ivar,i,0:nptsy)
          termy1(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy(0:nptsy))
          dg1dy(ivar,i,0:nptsy)=termy1(0:nptsy)
         END DO
       END DO

       split_y1(1:nvar,0:nptsx,0:nptsy)=dg1dy(1:nvar,0:nptsx,0:nptsy)

!********** Compute 2nd split flux term *********
       g2(0:nptsx,0:nptsy)=rho_v(0:nptsx,0:nptsy)

       DO i=0, nptsx
         termy2(0:nptsy)=g2(i,0:nptsy)
         termy3(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy2(0:nptsy))
         dg2dy(i,0:nptsy)=termy3(0:nptsy)
       END DO

       DO ivar=1, nvar
         split_y2(ivar,0:nptsx,0:nptsy)=phi(ivar,0:nptsx,0:nptsy)*dg2dy(0:nptsx,0:nptsy)
       END DO

!********** Compute 3rd split flux term *********
       DO ivar=1,nvar
         g3(ivar,0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy) ! rho*phi
       END DO

       DO ivar=1, nvar
         DO i=0, nptsx
          termy4(0:nptsy)=g3(ivar,i,0:nptsy)
          termy5(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy4(0:nptsy))
          dg3dy(ivar,i,0:nptsy)=termy5(0:nptsy)
         END DO
       END DO

       DO ivar=1, nvar
         split_y3(ivar,0:nptsx,0:nptsy)=uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*dg3dy(ivar,0:nptsx,0:nptsy) ! v*d
       END DO

!********** Compute 4th split flux term *********
       DO ivar=1,nvar
         g4(ivar,0:nptsx,0:nptsy)=uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy) ! v*phi
       END DO

       DO ivar=1, nvar
         DO i=0, nptsx
          termy6(0:nptsy)=g4(ivar,i,0:nptsy)
          termy7(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy6(0:nptsy))
          dg4dy(ivar,i,0:nptsy)=termy7(0:nptsy)
         END DO
       END DO

       DO ivar=1, nvar
         split_y4(ivar,0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*dg4dy(ivar,0:nptsx,0:nptsy) ! rho*dg4dy
       END DO

!********** Compute 5th split flux term *********
       DO ivar=1,nvar
         g5(ivar,0:nptsx,0:nptsy)=phi(ivar,0:nptsx,0:nptsy) ! phi
       END DO

       DO ivar=1, nvar
         DO i=0, nptsx
          termy8(0:nptsy)=g5(ivar,i,0:nptsy)
          termy9(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy8(0:nptsy))
          dg5dy(ivar,i,0:nptsy)=termy9(0:nptsy)
         END DO
       END DO

       DO ivar=1, nvar
         split_y5(ivar,0:nptsx,0:nptsy)=rho_v(0:nptsx,0:nptsy)*dg5dy(ivar,0:nptsx,0:nptsy) ! rho_v*dg5dy
       END DO

!********** Compute 6th split flux term *********
       g6(0:nptsx,0:nptsy)=uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy) ! v

       DO i=0, nptsx
          termy10(0:nptsy)=g6(i,0:nptsy)
          termy11(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy10(0:nptsy))
          dg6dy(i,0:nptsy)=termy11(0:nptsy)
       END DO

       DO ivar=1, nvar
         split_y6(ivar,0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy)*dg6dy(0:nptsx,0:nptsy) ! rho*ph
       END DO

!********** Compute 7th split flux term *********
        g7(0:nptsx,0:nptsy)=uold(1,0:nptsx,0:nptsy) ! rho

       DO i=0, nptsx
          termy12(0:nptsy)=g7(i,0:nptsy)
          termy13(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy12(0:nptsy))
```

```fortran
          dg7dy(i,0:nptsy)=termy13(0:nptsy)
       END DO

       DO ivar=1, nvar
         split_y7(ivar,0:nptsx,0:nptsy)=uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*phi(ivar,0:nptsx,0:nptsy) &
                                                                            *dg7dy(0:nptsx,0:nptsy) ! v*phi*dg7dy
       END DO

       dgdy= alpha_split*split_y1 + beta_split*(split_y2+split_y3+split_y4) + &
             (1-alpha_split-2*beta_split)*(split_y5+split_y6+split_y7)

    END SUBROUTINE glux


!************** Derivative of pressure terms with y *************
     SUBROUTINE dpre_dy
      INTEGER :: i, j, ivar, k
      REAL, DIMENSION(:) :: termy14(0:nptsy), termy15(0:nptsy)

     pre_mat_y(1,0:nptsx,0:nptsy)=0.0d0
     pre_mat_y(2,0:nptsx,0:nptsy)=0.0d0
     pre_mat_y(3,0:nptsx,0:nptsy)=pre(0:nptsx,0:nptsy)
     pre_mat_y(4,0:nptsx,0:nptsy)=(uold(3,0:nptsx,0:nptsy)/uold(1,0:nptsx,0:nptsy)*pre(0:nptsx,0:nptsy))*(1.0d0/(gam-1.0d0

      DO ivar=1, nvar
       DO i=0, nptsx
          termy14(0:nptsy)=pre_mat_y(ivar,i,0:nptsy)
          termy15(0:nptsy)=MATMUL(Dy1(0:nptsy,0:nptsy),termy14(0:nptsy))
          dpdy(ivar,i,0:nptsy)=termy15(0:nptsy)
       END DO
      END DO

    END SUBROUTINE dpre_dy


!************** Derivative of viscous flux wrt x *************
     SUBROUTINE dflux_dx_visc
      INTEGER :: i, j, ivar, k
      REAL, DIMENSION(:) :: termx(0:nptsx), termx1(0:nptsx)

      DO ivar=1, nvar
       DO j=0, nptsy
          termx(0:nptsx)=f_visc(ivar,0:nptsx,j)
          termx1(0:nptsx)=MATMUL(Dx1(0:nptsx,0:nptsx),termx(0:nptsx))
          dfdx_visc(ivar,0:nptsx,j)=termx1(0:nptsx)
       END DO
      END DO

    END SUBROUTINE dflux_dx_visc


!************** Derivative of viscous flux wrt y *************
     SUBROUTINE dglux_dy_visc
      INTEGER :: i, j, ivar, k, l
      REAL, DIMENSION(:) :: termy(0:nptsy), termy1(0:nptsy)

      DO ivar= 1, nvar
        DO i = 0, nptsx
          termy(0:nptsy) = g_visc(ivar, i, 0:nptsy)
          termy1(0:nptsy) = MATMUL(Dy1(0:nptsy, 0:nptsy), termy(0:nptsy))
          dgdy_visc(ivar, i, 0:nptsy) = termy1(0:nptsy)
        END DO
      END DO

    END SUBROUTINE dglux_dy_visc


!************** SAT Left (left on x axis) *************
    SUBROUTINE SAT_x_left
      INTEGER :: i, j, ivar, k
      REAL(DP) :: sigma
      ALLOCATE(temp1d(1:nvar,0:nptsx,0:nptsy),correction(1:nvar,0:nptsy))

       sigma=1.7d0 ! Energy stable for sigma>=0.5

      ! Target values at left boundary
      ! Perioidc BC; so left boundary=right, u_0=u_n
        g_x_0(1,0:nptsy)=1.0d0
        g_x_0(2,0:nptsy)=0.0d0
        g_x_0(3,0:nptsy)=0.0d0
        g_x_0(4,0:nptsy)=2.5d0

      hinv_x0=Hinvx(0,0)

      correction(1:nvar,0:nptsy)=sigma*hinv_x0*(uold(1:nvar,0,0:nptsy)-g_x_0(1:nvar,0:nptsy))

      ! Intializing SAT_x0 as a (nx X ny) matrix of zeros
      SAT_x0(1:nvar,0:nptsx,0:nptsy)=0.0d0
      ! Replacing the left-hand side of the matrix (i.e., x=0 of all rows) with SAT_x0 values; the remaining values are zer

      DO j=0, nptsy
        SAT_x0(1:nvar,0,j)=correction(1:nvar,j)
      END DO


     DEALLOCATE(temp1d,correction)

    END SUBROUTINE SAT_x_left


!************** SAT Right (right on x axis) *************
    SUBROUTINE SAT_x_right
      INTEGER :: i, j, ivar, k
      REAL(DP) :: sigma
      ALLOCATE(temp1d(1:nvar,0:nptsx,0:nptsy),correction(1:nvar,0:nptsy))

       sigma=1.7d0   ! Energy stable for sigma>=0.5

       ! Target values at right boundary
      ! Perioidc BC; so right boundary=left, u_0=u_n
        g_x_n(1,0:nptsy)=1.0d0
        g_x_n(2,0:nptsy)=0.0d0
        g_x_n(3,0:nptsy)=0.0d0
        g_x_n(4,0:nptsy)=2.5d0

     hinv_xn=Hinvx(nptsx,nptsx)

      correction(1:nvar,0:nptsy)=sigma*hinv_xn*(uold(1:nvar,nptsx,0:nptsy)-g_x_n(1:nvar,0:nptsy))

      ! Intializing SAT_xn as a (nx X ny) matrix of zeros
      SAT_xn(1:nvar,0:nptsx,0:nptsy)=0.0d0
      ! Replacing the right-hand side of the matrix (i.e., x=nptsx of all rows) with SAT_xn values; the remaining values ar
      DO j=0, nptsy
        SAT_xn(1:nvar,nptsx,j)=correction(1:nvar,j)
      END DO

    DEALLOCATE(temp1d,correction)

    END SUBROUTINE SAT_x_right


!************** SAT Bottom (on Y axis) *************
    SUBROUTINE SAT_y_bottom
      INTEGER :: i, j, k
      REAL(DP) :: sigma
      REAL, DIMENSION(:) :: termy(0:nptsy), termy1(0:nptsy)
      ALLOCATE(temp1d(1:nvar,0:nptsx,0:nptsy),correction(1:nvar,0:nptsx))

       sigma=1.7d0 ! Energy stable for sigma>=0.5

      ! Target value at bottom boundary
        g_y_0(1,0:nptsx)=1.0d0
        g_y_0(2,0:nptsx)=0.0d0
        g_y_0(3,0:nptsx)=0.0d0
        g_y_0(4,0:nptsx)=2.5d0

      hinv_y0=Hinvy(0,0)

      correction(1:nvar,0:nptsx)=sigma*hinv_y0*(uold(1:nvar,0:nptsx,0)-g_y_0(1:nvar,0:nptsx))

      ! Intializing SAT_y0 as a (nx X ny) matrix of zeros
      SAT_y0(1:nvar,0:nptsx,0:nptsy)=0.0d0
      ! Replacing the top side of the matrix (i.e., y=0 of all rows) with SAT_y0 values; the remaining values are zeros
      DO i=0, nptsx
        SAT_y0(1:nvar,i,0)=correction(1:nvar,i)
      END DO

      DEALLOCATE(temp1d,correction)

    END SUBROUTINE SAT_y_bottom


!************** SAT Top(on Y axis) *************
    SUBROUTINE SAT_y_top
      INTEGER :: i, j, k
      REAL(DP) :: sigma
      REAL, DIMENSION(:) :: termy(0:nptsy), termy1(0:nptsy)
      ALLOCATE(temp1d(1:nvar,0:nptsx,0:nptsy),correction(1:nvar,0:nptsx))

          sigma=0.5d0  ! Energy stable for sigma>=0.5


      ! Target value at top boundary
        g_y_n(1,0:nptsx)=1.0d0
        g_y_n(2,0:nptsx)=1.0d0
        g_y_n(3,0:nptsx)=0.0d0
        g_y_n(4,0:nptsx)=3.0d0

      hinv_yn=Hinvy(nptsy,nptsy)

      correction(1:nvar,0:nptsx)=sigma*hinv_yn*(uold(1:nvar,0:nptsx,nptsy)-g_y_n(1:nvar,0:nptsx))

      ! Intializing SAT_y0 as a (nx X ny) matrix of zeros
      SAT_yn(1:nvar,0:nptsx,0:nptsy)=0.0d0
      ! Replacing the bottom side of the matrix (i.e., y=nptsy of all rows) with SAT_yn values; the remaining values are ze
      DO i=0, nptsx
        SAT_yn(1:nvar,i,nptsy)=correction(1:nvar,i)
      END DO

      DEALLOCATE(temp1d,correction)

    END SUBROUTINE SAT_y_top


!************** RHS *************
    SUBROUTINE right_hand_side

     ! Inviscid
     CALL flux
     CALL dpre_dx
     CALL glux
     CALL dpre_dy

     ! Viscous
     CALL flux_visc
     CALL dflux_dx_visc
     CALL dglux_dy_visc
     ! SATs
     CALL SAT_x_left
     CALL SAT_x_right
     CALL SAT_y_bottom
     CALL SAT_y_top

     rhs(1:nvar,0:nptsx,0:nptsy)=-dfdx(1:nvar,0:nptsx,0:nptsy)-dpdx(1:nvar,0:nptsx,0:nptsy) +&
                                  dfdx_visc(1:nvar,0:nptsx,0:nptsy)- &
                                  dgdy(1:nvar,0:nptsx,0:nptsy)-dpdy(1:nvar,0:nptsx,0:nptsy) + &
                                  dgdy_visc(1:nvar,0:nptsx,0:nptsy)- &
                                  SAT_x0(1:nvar,0:nptsx,0:nptsy)-SAT_xn(1:nvar,0:nptsx,0:nptsy)- &
                                  SAT_y0(1:nvar,0:nptsx,0:nptsy)-SAT_yn(1:nvar,0:nptsx,0:nptsy)

    END SUBROUTINE right_hand_side


!************** Decompose the solution vector back to primitive variables after each dt *************
! This also computes other essential quantities for visualization.
! This should be called inside RK-4 (after computing uold at each dt)since this decomposes pressure
! from the solution vector which will then be needed to compute the flux in the next dt.
    SUBROUTINE decomp(ut)
```

```fortran
      INTEGER :: i,j
      REAL(DP), DIMENSION(:, 0:, 0:) :: ut

      DO i=0, nptsx
        DO j=0, nptsy
          rho(i,j)=ut(1,i,j)
          uvel(i,j)=ut(2,i,j)/ut(1,i,j)
          vvel(i,j)=ut(3,i,j)/ut(1,i,j)
          et(i,j)=ut(4,i,j)/ut(1,i,j)
          ke(i,j)=0.5d0*(uvel(i,j)**2.0d0+vvel(i,j)**2.0d0)
          temp(i,j)=(gam-1.0d0)*(et(i,j)-ke(i,j))/rgas
          pre(i,j)=rho(i,j)*rgas*temp(i,j)
          sos(i,j)=SQRT(gam*pre(i,j)/rho(i,j)) ! Speed of sound
          mach(i,j)=uvel(i,j)/sos(i,j)
        END DO
      END DO


    OPEN(unit = 2, file = 'sol.dat', status = 'replace')
    WRITE(2,'(A)') 'VARIABLES = "X", "Y", "rho", "uvel", "vvel", "pre"'
      WRITE(2,*) 'ZONE I=',nptsx+1,', J=',nptsy+1,', ZONETYPE=ORDERED,'
    WRITE(2,*) 'DATAPACKING=POINT, SOLUTIONTIME=0.0'
    DO i = 0, nptsx
      DO j=0, nptsy
     ! WRITE(2, *) x(k), u(1,k), u(2,k)
       WRITE(2, *) x(i), x(j), rho(i,j), uvel(i,j), vvel(i,j), pre(i,j)
      END DO
    END DO
    CLOSE(2)

    END SUBROUTINE decomp


!************** Time integration *************
  ! RK-4
  SUBROUTINE rk4
    USE types_vars
    USE variables

    IMPLICIT NONE
    INTEGER :: i,j, k

    uold=u
      OPEN(unit = 44, file = 'convergence.dat', status = 'replace')
    DO j=0, nptst

!********************************************* STEP 1 *********************************************
       CALL right_hand_side
       k1=Dt*rhs
       uold=u+(k1/2.0d0)
       CALL decomp(uold)

!********************************************* STEP 2 *********************************************
       CALL right_hand_side
       k2= Dt*rhs
       uold=u+(k2/2.0d0)
       CALL decomp(uold)

!********************************************* STEP 3 *********************************************
       CALL right_hand_side
       k3=Dt*rhs
       uold=u+k3
       CALL decomp(uold)

!********************************************* STEP 4 *********************************************
       CALL right_hand_side
       k4=Dt*rhs
       uold=u+((1.0d0/6.0d0)*(k1+(2.0d0*k2)+(2.0d0*k3)+k4))
       CALL decomp(uold)
       u=uold


       t=t+Dt
       WRITE(*,*) 't=',t, ', Max of u1=', MAXVAL(u(1,:,:)), &
       ', Max of u2=', MAXVAL(u(2,:,:)), ', Max of u3=', MAXVAL(u(3,:,:)), &
              ', Max of u4=', MAXVAL(u(4,:,:))
       WRITE(*,*) 't=',t, ', Min of u1=', MINVAL(u(1,:,:)), ', Min of u2=', MINVAL(u(2,:,:)), &
       ', Min of u3=', MINVAL(u(3,:,:)), &
                           ', Min of u4=', MINVAL(u(4,:,:))
      WRITE(44,*) SUM(u(2,:,:))


    END DO
      CLOSE(44)
  END SUBROUTINE rk4

  END MODULE subroutines



  !*************************************************************************
  !*************************************************************************
  !*************************************************************************
  !*************************************************************************
  ! MAIN PROGRAM
  ! Program 2-D CNSE SBP-SAT Split-form
  PROGRAM CNSE2d_Split_form_lid_cavity
    USE types_vars
    USE variables
    USE subroutines
    INTEGER :: t1, rate, t2

    CALL system_clock(t1, rate)

    CALL inputs
    CALL memalloc

    CALL grid2d
    CALL init2d

    CALL solvec
    CALL decomp(u)

    CALL matrix(1,nptsx)
    Dx1=D1mat
    Hinvx=Hmatinv
 !   DEALLOCATE(Hmatinv,Qmat,Q_tran,D1mat,D2mat)
    DEALLOCATE(Hmatinv,Qmat,D1mat,D2mat,Mmat,Smat,Bmat)

    CALL matrix(2,nptsy)
    Dy1=D1mat
    Hinvy=Hmatinv
 !   DEALLOCATE(Hmatinv,Qmat,Q_tran,D1mat,D2mat)
    DEALLOCATE(Hmatinv,Qmat,D1mat,D2mat,Mmat,Smat,Bmat)

    CALL rk4

    WRITE(*,*) 'Nx=',nptsx,', Ny=',nptsy, ', CFL=',cfl_ip
    WRITE(*,*) 'Dx=', Dx, ', Dy=', Dy, ', Dt=', Dt

    CALL dealloc

    CALL system_clock(t2)
    WRITE(*,*) 'Elapsed time=', REAL(t2 - t1)/REAL(rate), 's'

  END PROGRAM CNSE2d_Split_form_lid_cavity
```