Homework-5 of CFD Course: 038782
Lecturer: Prof.Steven Frankel

Ronith Stanly

Time spent: 10 Jan, 2019 to 19 Jan, 2019

Flux Reconstruction Method

[Huynh(2007)]

1-D Linear and non-linear scalar advection

Q: Write a modern Fortran program to implement the flux-reconstruction (FR) scheme for the 1-D scalar conservation law: $\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0$; for $f = u$ and $u(x, 0) = \exp(-20x^2)$ (i.e., Gaussian profile). Repeat above for Burgers equation with $f = u^2/2$ and $u(x, 0) = sin(x)$. Perform a careful study of error versus N and P; e.g. h-p refinement.

Algorithm and checks at different levels:

1. **Domain discretization**

   Discretize the domain into n elements, bounded by $n + 1$ faces ($nels = n,\ nfaces = n + 1$).

   ```
   CALL grid1d
   ```

2. **Assign solution points**

   Within each element, the solution (which is continuous within an element, while discontinuous between adjacent elements) is represented by a polynomial of degree p using $p+1$ points. The location of these points within each element is decided by the choice of the family of points Gauss-Legendre (solution points which do not include faces bounding each element) or Gauss-Lobatto (solution points which include faces bounding each element).
   For each family of points, the location of these $p+1$ points are available as standard values on a standard domain/element which extends from $(-1, +1)$. So, to find the location (x) of the $p + 1$ solution points within each element of our domain of interest, each element is mapped onto the standard element by inputting the location of the faces bounding each element (x_n and x_{n+1}) into:

   $$x = \Gamma^{-1}(r) = \left(\frac{1-r}{2}\right) x_n + \left(\frac{1+r}{1}\right) x_{n+1} \tag{1}$$

   where r values are the position of the solution points on the standard element. (Vincent *et al.*(2011))
   For example, for a $p = 2$ degree polynomial, the $p + 1 = 3$ solution points on the standard element are $r(1) = -0.774597, r(2) = 0.0, r(3) = +0.774597$.

   The following function assigns r values based on the chosen value of p.

   ```
   CALL glnodes
   ```

3. **Initialization**

$$u(x,0) = exp(-20x^2) \tag{2}$$

The following function computes the internal solution points within all elements using Eq.(1) and assigns the initial value of solution variable u (Eq.(2)) at those internal solution points (the values are not assigned to the points on the faces of each element because we are using Gauss-Legendre points and also because the solution will be discontinuous at the faces).

```
CALL init1d
```

4. **Element $\leftrightarrow$ Face mapping**

Given the array index of an element, it is essential to know the array index of its neighbouring faces. Its vice-versa is also required. This is because certain arrays like the array containing the discontinuous flux (to be discussed) within each element will run through the elements ($nels = n$), while certain others like the array containing the interaction flux (to be discussed) will be indexed through the faces ($nfaces = n+1$); and it will be required to use them together in an equation, the loop variable for which will be either running throguh $nels$ or $nfaces$. So a face to element and element to face mapping is used, which is further used to enforce periodic boundary condition that is required for the current set of test cases.

```
CALL face_to_element_map
CALL element_to_face_map
```

5. **Construct lagrange polynomials**

The following function computes the lagrange basis functions at each solution point within the standard element as shown in Eq.(3). The function computes values based on the inputted value of the location to which the constructed polynomial should then extrapolate a variable. So, this function is called twice to calculate the lagrange basis function values for an input location of -1 (left face of standard element) and +1 (right face of standard element).

$$l_{i=1}^{p+1} = \prod_{j=1, j\neq i}^{p+1} \left(\frac{r - r_j}{r_i - r_j} \right) \tag{3}$$

```
CALL lagrangePoly(-1.0d0,l_l)
CALL lagrangePoly(1.0d0,l_r)
```

!!! Check point 1: To ensure that you have correctly coded up the above function, check whether the sum of all the basis functions (i.e., individual sums of the arrays l_l and l_r) amount to one. This is because later in the code when the solution value and the flux value will have to be extrapolated to the element faces using these basis function values, these values end-up being coefficients in that equation, and hence for consistency the coefficients should sum-up to one. And these lagrange polynomials are designed in such a way that they add-up to one; however, if your code for the function has an error, then you may not get a sum of one.

6. **Compute derivative of lagrange polynomial basis functions**

To compute the right-hand side of the equation, the derivative of the lagrange basis functions of each solution point with respect to each solution point itself is required. It is computed as shown below in Eq.(4) (The equation was found here). The function first computes the lagrange basis functions and then its derivatives. This function returns a $(p+1) \times (p+1)$ matrix because there are $(p+1)$ basis functions and $(p+1)$ solution points within the standard element.

$$l' = \frac{dl}{dr} \implies l'_{i=1}^{p+1} = \sum_{l=1, l\neq i}^{p+1} \frac{1}{r_i - r_l} \underbrace{\prod_{j=1, j\neq (i,l)}^{p+1} \left(\frac{r - r_j}{r_i - r_j} \right)}_{l_i} \tag{4}$$

```
CALL lag_poly_der
```

!!! Check point 2: To ensure that you have correctly coded up the above function, check whether the sum of all the terms in each row (i.e., sum of derivatives of one basis function taken about $p+1$ points) amount to zero. This is because, since as mentioned in the previous check point, the sum of basis functions yield the coefficient which is a constant; hence, its derivative should be zero.

7. **Time marching**

Three stage Strongly Stability Preserving Runge-Kutta scheme (RK3SSP) is used here. It was observed that while Euler time-stepping was employed, the solution value grew to unreasonable values very quickly.

```
CALL rk3
```

(a) **Compute flux**

Within each element the flux (which is discontinuous between elements) is computed at the solution points. It is then normalized by the Jacobian (which is in-effect like normalizing by the average grid size) computed at the element faces on the domain of interest (and not on standard element).

$$J = \frac{x_{n+1} - x_n}{2} \tag{5}$$

$$f = \frac{(a \cdot u(x,t))}{J} \tag{6}$$

```
CALL fluxD
```

(b) **Extrapolating flux to the element faces**

In order to be able to compute the derivative of the flux so as to solve the equation, it should be made continuous between elements. The first steps towards it is to extrapolate the flux from the interior solution points to the element faces, so as to get some value of the flux at the element faces. This is done, as shown below, by using the lagrange cardinal/basis functions computed earlier. The same is performed for the solution variable.

$$\hat{u}^{\delta} = \sum_{i=1}^{p+1} \hat{u}^{\delta} l_i \tag{7}$$

$$\hat{f}^{\delta D} = \sum_{i=1}^{p+1} \hat{f}^{\delta D} l_i \tag{8}$$

```
CALL extrapolate
```

(c) **The Riemann problem**

After extrapolating the flux to the element faces, each face will have two flux values that are discontinuous (one on the left of the face and one of the right of the face). A Riemann solver is used to compute a single flux at each face. In the present example, Roe's Riemann solver was used. (Huynh(2007))

$$a(\tilde{u}) = \begin{cases} \frac{f(u_R)-f(u_L)}{u_R-u_L}, & \text{if } u_L \neq u_R \\ a(u_L) = a(u_R), & \text{otherwise} \end{cases}$$

$$f_{face} = \frac{1}{2} \cdot [f(u_L) + f(u_R)] - \frac{1}{2}|a(\tilde{u})|(u_R - u_L) \tag{9}$$

```
CALL interaction_flux
```

(d) **Correction function and its derivative**

The flux which is discontinuous between elements can be made continuous by using correction functions which do not alter the flux within the element, but will bend it at the faces in such a way that the flux at the faces matches with the common flux value at the faces between two elements calculated using the Riemann solver. For this, two Legendre polynomials should be constructed (one for left face and one for right face). This

process of calculating the legendre polynomial should ideally be a part of the code, if it has to be capable of handling polynomials of arbitrary degree. However, for the present test cases, expressions for the derivatives of polynomials (and not the expressions for the legendre polynomial itself) upto ninth degree were hard-coded. Only the derivatives of the legendre polynomial are used for the computation of the right-hand side.

```
CALL corr_fn_der
```

(e) **Divergence of flux: RHS**

$$\frac{\partial \hat{u}^{\delta}}{\partial t} + \frac{\partial \hat{f}^{\delta}}{\partial r} = 0 \tag{10}$$

$$\implies \frac{\partial \hat{u_i}^{\delta}}{\partial t} = \underbrace{-\frac{\partial \hat{f}^{\delta}(r_i)}{\partial r}}_{RHS} \tag{11}$$

Within each element, the divergence of flux is calculated at each solution point using:

$$\frac{\partial \hat{f}^{\delta}(r_i)}{\partial r} = \underbrace{\sum_{j=1}^{p+1} \underbrace{\hat{f}_j^{\delta D}}_{V} \underbrace{\frac{dl_j}{dr}(r_i)}_{M}}_{V-M\ multiplication} + \left(\underbrace{\hat{f}_L^{\delta I}}_{S} - \underbrace{\hat{f}_L^{\delta D}}_{S}\right) \underbrace{\frac{dg_L}{dr}(r_i)}_{V} + \left(\underbrace{\hat{f}_R^{\delta I}}_{S} - \underbrace{\hat{f}_R^{\delta D}}_{S}\right) \underbrace{\frac{dg_R}{dr}(r_i)}_{V} \tag{12}$$

where: V =Vector, M =Matrix and S =Scalar

The different independent terms are as follows:

- $\hat{f}_j^{\delta D}$ =Discontinuous flux calculated at interior/solution points of the present element
- $\frac{dl_j}{dr}(r_i)$ =Matrix containing derivatives of the lagrange polynomial basis functions
- $\hat{f}_L^{\delta I}, \hat{f}_R^{\delta I}$ =Interaction flux calculated at the left and right faces of the present element using the Riemann solver
- $\hat{f}_L^{\delta D}, \hat{f}_R^{\delta D}$ =Discontinuous flux extrapolated to the left and right faces of the present element
- $\frac{dg_L}{dr}(r_i), \frac{dg_R}{dr}(r_i)$ =Derivatives of the correction functions on the left and right faces of the present element

```
CALL rhs
```

The first term on the RHS is vector-matrix multiplication, which yields a vector as output and should be implemented carefully. The following example with $p+1 = 3$ was considered while implementing it in the code:

$$\underbrace{\sum_{j=1}^{p+1} \underbrace{\hat{f}_j^{\delta D}}_{V} \underbrace{\frac{dl_j}{dr}(r_i)}_{M}}_{} = \begin{bmatrix} f_1 & f_2 & f_3 \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l_1}{\partial r_1} & \frac{\partial l_1}{\partial r_2} & \frac{\partial l_1}{\partial r_3} \\ \frac{\partial l_2}{\partial r_1} & \frac{\partial l_2}{\partial r_2} & \frac{\partial l_2}{\partial r_3} \\ \frac{\partial l_3}{\partial r_1} & \frac{\partial l_3}{\partial r_2} & \frac{\partial l_3}{\partial r_3} \end{bmatrix} = \begin{bmatrix} f_1 \frac{\partial l_1}{\partial r_1} + f_2 \frac{\partial l_1}{\partial r_2} + f_3 \frac{\partial l_1}{\partial r_3} \\ f_1 \frac{\partial l_2}{\partial r_1} + f_2 \frac{\partial l_2}{\partial r_2} + f_3 \frac{\partial l_2}{\partial r_3} \\ f_1 \frac{\partial l_3}{\partial r_1} + f_2 \frac{\partial l_3}{\partial r_2} + f_3 \frac{\partial l_3}{\partial r_3} \end{bmatrix} \tag{13}$$

This was implemented as:

```
rhs1(:,:)=0.d0
DO i=1, nels
  DO ppp1=1, pp1
    DO k=1, pp1
      rhs1(i,ppp1)=rhs1(i,ppp1)+f(i,k)*lpdm(ppp1,k)
    END DO
  END DO
END DO
```

Stability

To maintain L_2 stability CFL condition is usually chossen as:

$$|a|\frac{\Delta t}{\Delta x} \leq CFL \tag{14}$$

For DG discretizations, since we are using polynomials of degree p (which takes $p+1$ points within an element of size Δx) and a $p+1$ stage RK method of order $p+1$ (which give rise to an $(p+1)^{th}$-order accurate method), it is recommended to take

$$CFL = \frac{1}{2p+1} \tag{15}$$

in practice (Bernado and Shu(2001)).

To ensure stability for all the different cases that were studied for *hp*-refinement, a standard/constant CFL was used for all test cases, which was much lower than the recommended values (as shown in the table given below) that were obtained following the suggestion of Bernado and Shu(2001). Recommended and used CFL values for the cases studied here for *hp*-refinement are as follows:

	CFL condition	
p	**Recommended**	**Used here**
2	0.2	0.01
4	0.11	0.01
6	0.077	0.01

Test case 1: Linear advection

Linear flux function, $f = a \cdot u(x,t), a = 1$

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \tag{16}$$

$$\frac{\partial u}{\partial t} + \underbrace{\frac{df(u)}{du}}_{Wave\ speed, a=1} \cdot \frac{\partial u(x,t)}{\partial x} = 0 \tag{17}$$

$$\implies \frac{\partial u}{\partial t} + \frac{\partial u(x,t)}{\partial x} = 0 \tag{18}$$

$\implies f = u(x,t)$ and $a = 1$

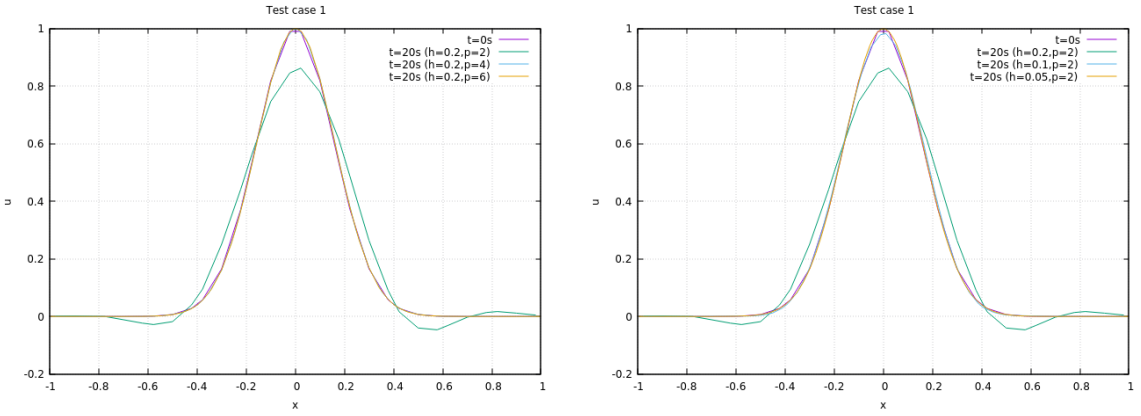
Domain=[-1,1]
Time=[0,20]
Periodic boundary condition and Gaussian initial profile given by:

$$u(x,0) = exp(-20x^2) \tag{19}$$

Time stepping by RK3-SSP.
Gauss-Legendre solution points.

Grid size (h) |	**Polynomial (p)** |	L_∞ **error**	L_1 **error**	L_2 **error**
0.2	2	4.81834 E -3	4.09035 E -2	1.04269 E -2
0.2	4	1.77587 E -4	1.98601 E -3	4.74869 E -4
0.2	6	1.08846 E -4	1.94193 E -3	3.97776 E -4
0.2	2	4.81834 E -3	4.09035 E -2	1.04269 E -2
0.1	2	3.16551 E -4	4.00538 E -3	8.50836 E -4
0.05	2	2.11981 E -5	5.26760 E -4	8.17771 E -5

It can be seen that, for a given h, the solution accuracy improves with refinement in p. The vice-versa is also true. One-time refinement of either of these, i.e., a grid of 0.2 size with with a polynomial of degree 4 and a grid of size 0.1 with a polynomial of degree 2, removes the oscillations in the solution that was present in the solution for the case with a coaser value for either h or p.



(a) p-refinement (b) h-refinement

Figure 1: Test Case 1 : Linear advection- Gaussian profile

Test case 2: Non-Linear advection (Inviscid Burger's)

Non-linear flux function, $f = \frac{u^2}{2}$

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

$$\frac{\partial u}{\partial t} + \underbrace{\frac{df(u)}{du}}_{Wave\ speed, a(u)=u(x,t)} \cdot \frac{\partial u(x,t)}{\partial x} = 0 \tag{21}$$

$$\implies \frac{\partial u}{\partial t} + u(x,t) \cdot \frac{\partial u(x,t)}{\partial x} = 0 \tag{22}$$

$\implies f = u(x,t)$ and $a(u) = u(x,t)$

Domain=[0,2]
Time=[0,0.4]
Periodic boundary condition and sinusoidal initial profile given by:

$$u(x,0) = sin(\pi x) \tag{23}$$

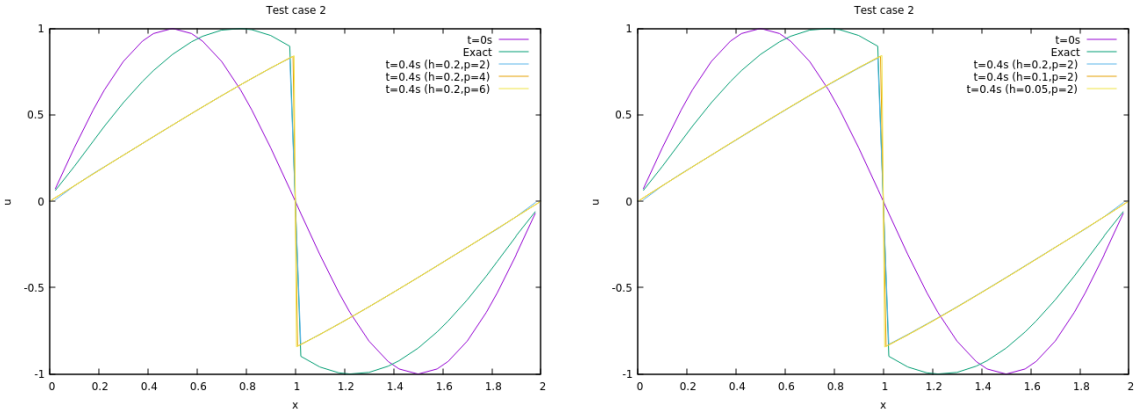
Time stepping by RK3-SSP.
Gauss-Legendre solution points.

Grid size (h)	**Polynomial (p)**	L_∞ **error**	L_1 **error**	L_2 **error**
0.2	2	1.38425 E -2	2.74126 E -1	5.47771 E -2
0.2	4	8.31168 E -3	2.71934 E -1	4.24200 E -2
0.2	6	5.93530 E -3	2.71688 E -1	3.58492 E -2
0.2	2	1.38425 E -2	2.74126 E -1	5.47771 E -2
0.1	2	6.89085 E -3	2.71959 E -1	3.85388 E -2
0.05	2	3.44560 E -3	2.71917 E -1	2.72414 E -2

All the h and p values seem to predict the shock location with reasonable accuracy. However, one initial refinement of either h or p, from the baseline case, gave better prediction of the shock location.

Test case 3 and validation: Non-Linear advection (Inviscid Burger's)

To check the correctness of implementation of the FRM solver and its robustness for the non-linear case, a test case found in literature [N. Petrovskaya(2006)] was simulated and the results were



(a) p-refinement (b) h-refinement

Figure 2: Test Case 2 : Non-linear advection- Burger's equation

compared for N=16 and p=7 as shown below in Figure (3). The errors in that case were also studied for hp-refinement.

Domain=[0,1]
Time=[0,0.15]
Periodic boundary condition and sinusoidal initial profile given by:

$$u(x,0) = 0.25 + 0.5\ sin(\pi(2x-1)) \tag{24}$$

Time stepping by RK3-SSP.
Gauss-Legendre solution points.

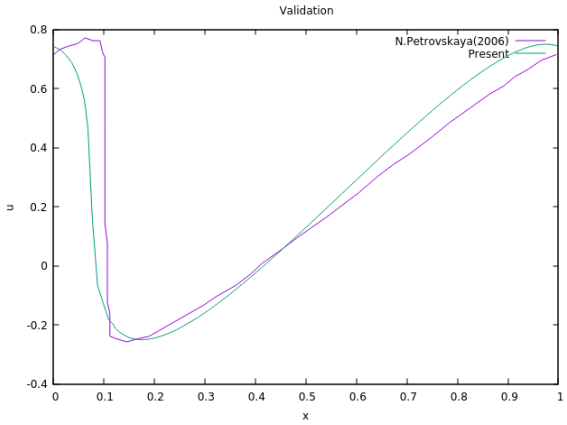
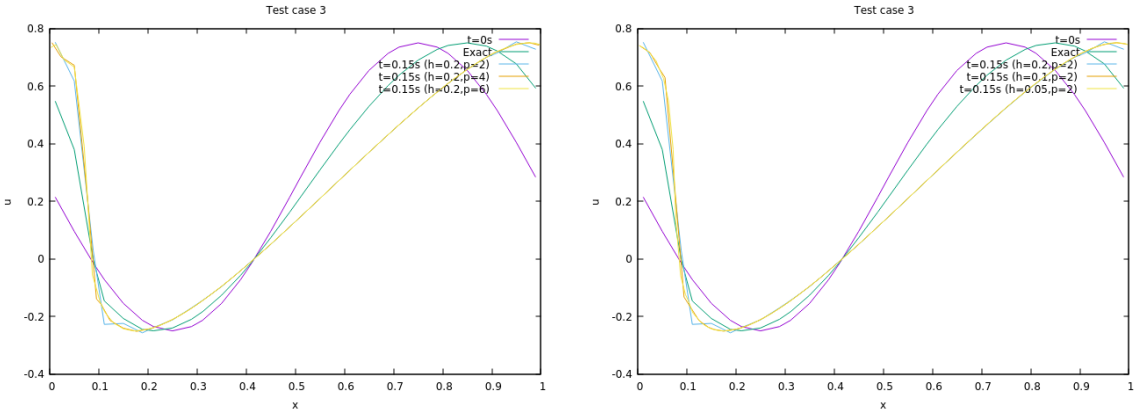


Figure 3: Validation

The current implementation of the FRM solver is able to predict the solution of the non-linear advection equation (inviscid Burger's) reasonably well as compared to the test case from literature. The variations between them might be due to the difference in Riemann solver employed in both cases.

Grid size (h)	**Polynomial** (p)	L_∞ **error**	L_1 **error**	L_2 **error**
0.2	2	7.94208 E -3	8.53779 E -2	1.97378 E -2
0.2	4	5.36095 E -3	8.53594 E -2	1.52475 E -2
0.2	6	3.76826 E -3	8.45711 E -2	1.28590 E -2
0.2	2	7.94208 E -3	8.53779 E -2	1.97378 E -2
0.1	2	4.34006 E -3	8.67627 E -2	1.41089 E -2
0.05	2	2.07342 E -3	8.60014 E -2	9.92116 E -3

This test case clearly shows the variation in the solution accuracy with each subsequent refinement of either h or p. The removal of oscillation with increasing refinements is to be noted.



(a) p-refinement (b) h-refinement

Figure 4: Test Case 3 : Non-linear advection- Burger's equation
Test case from N. Petrovskaya(2006)

Flux Reconstruction code

```
!******************************************************************************
!******************************************************************************
!!!!****************** Flux Reconstruction Method **********************!!!!
! This code solves 1-D scalar linear and non-linear inviscid advection
! equation and employs the FRM proposed by Huynh(2007).
! Author: Ronith Stanly
! Last updated: 19 January, 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    = 3.14159265358979323846264338327950288419 7_dp
  REAL(DP), PARAMETER :: pio2  = 1.57079632679489661923132169163975144209858_dp
  REAL(DP), PARAMETER :: twopi = 6.28318530717958647692528676655900576839 4_dp
END MODULE types_vars

!******************************************************************************
!******************************************************************************
!******************************************************************************
!******************************************************************************
! This module defines and allocates the variables needed in the current simulation
! If needed, add new variables at teh beginning of the module, then allocate
! them in the subroutine memalloc
MODULE variables
```

```
    USE types_vars
    ! Add new variables here
    INTEGER :: nels, ntimes, nfaces, nptst, iwave, tick, p, pp1, ispeed
    REAL(DP) :: a, b, Dx, t, cfl, cfl_ip, u_rk3, u_rk4
    REAL(DP) :: c, d, Dt, J, l2_rk3, l1_rk3, l_infty_rk3
    ! 1-D arrays
!   INTEGER, ALLOCATABLE, DIMENSION(:) ::au
    REAL(DP), ALLOCATABLE, DIMENSION(:) :: x, time, dfdx, u_icc, au
    REAL(DP), ALLOCATABLE, DIMENSION(:) :: xgl, l_l, l_r, f_interaction, dlpdr, dlpp1dr, dgl, dgr
    ! 2-D arrays
    INTEGER, ALLOCATABLE, DIMENSION(:,:) :: f2e, e2f
    REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: x_internal, u, uold, f,u_face, f_face, lpdm, df, rhs1
    REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: k1, k2, k3, k4, u_iee, u_dd, auu, a_ini, u_ic

    CONTAINS

    ! Subroutine memalloc
    ! Allocation of the memory for the different arrays
    SUBROUTINE memalloc
      ! 1-D arrays
      ! Allocate memory for grid, solution, and flux function
      ALLOCATE(x(0:nfaces))
      ALLOCATE(dfdx(-1:nfaces+1))
      ALLOCATE(xgl(1:pp1),l_l(1:pp1),l_r(1:pp1),au(1:nfaces),f_interaction(1:nfaces))
      ALLOCATE( dlpdr(1:pp1),dlpp1dr(1:pp1),dgl(1:pp1),dgr(1:pp1))
      ! 2-D arrays
      ALLOCATE(f2e(1:nfaces,1:2),e2f(1:nels,1:2),x_internal(1:nels,1:pp1),u(1:nels,1:pp1),f(1:nels,1:pp1))
      ALLOCATE(u_face(1:nels,1:2),f_face(1:nels,1:2),lpdm(1:pp1,1:pp1),df(1:nels,1:pp1),rhs1(1:nels,1:pp1))
      ALLOCATE(uold(1:nels,1:pp1),k1(1:nels,1:pp1), k2(1:nels,1:pp1), k3(1:nels,1:pp1), k4(1:nels,1:pp1))
      ALLOCATE(auu(1:nels,1:pp1),u_iee(1:nels,1:pp1),u_dd(1:nels,1:pp1),a_ini(1:nels,1:pp1),u_ic(1:nels,1:pp1))
    END SUBROUTINE memalloc

    ! Subroutine dealloc
    ! Deallocation of the memory (end of the program)
    SUBROUTINE dealloc
      ! Deallocate memory for grid, solution, and flux function
      DEALLOCATE(x,dfdx,xgl,l_l,l_r,au,f_interaction,dlpdr,dlpp1dr,dgl,dgr,u,f,uold,k1,k2,k3,k4)
      DEALLOCATE(f2e,e2f,x_internal,u_face,f_face,lpdm,df,rhs1,auu,u_iee,u_dd,a_ini,u_ic)
    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

    ! Read from screen input information for program
    WRITE(*,*) '**** Flux Reconstruction Method of Huynh(2007) ****'
    WRITE(*,*) 'Please input the number of elements:'
    READ(*,*) nels
    nfaces = nels+1 !(nels) elements => (nels+1) points

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

    !WRITE(*,*) 'Time-step size=', Dt
    WRITE(*,*) 'Domain will be discretized with ', nfaces, ' points in space and ',nptst,' points in time'
    WRITE(*,*) 'for the given CFL number=', cfl_ip

    ! Echo print your input to make sure it is correct
    WRITE(*,*) 'Your 1D domain is from ', a, ' to ', b, 'and time is from ',c,'s to ', d,'s'
```

```
    WRITE(*,*) 'Specify Linear or non-linear flux function'
    WRITE(*,*) '...enter 1 for linear, or 0 for non-linear'
    READ(*,*) iwave

    WRITE(*,*) 'Specify initial condition'
    WRITE(*,*) '...enter 1 for u(x,0)=exp(-20*x^2), or 0 for u(x,0)=sin(pi*x)'
    READ(*,*) ispeed

    WRITE(*,*) 'Please input the degree of the polynomial representing solution variable'
    WRITE(*,*) '....available upto 9th order (i.e., enter 1,2,..,9)'
    WRITE(*,*) '....No. of points that will be used is one plus this number'
    WRITE(*,*) '....Enter degree of polynomial (p)'
    READ(*,*) p
    pp1=p+1

    ! Assume 1D domain is x[a,b] and time, t[c,d]
    IF (iwave==1) THEN ! Linear
      a = -1.0d0; b = 1.0d0; c = 0.0d0; d = 20.0d0
    ELSE ! Non-linear
      a = 0.0d0; b = 2.0d0; c = 0.0d0; d = 0.4d0
    END IF

    END SUBROUTINE inputs


!************ Structure of FRM elements and faces ***************

! Faces (showing 1 interior node):
! 1         2         3         4         5
! |----X----|----X----|----X----|----X----|
! Elements:
!     (1)       (2)       (3)       (4)


!************ Gauss-Legendre points within the standard element ***************
    SUBROUTINE glnodes
    IMPLICIT NONE

    !xgl(pp1)
    IF (p==0) THEN
      xgl(1)= 0.d0
    END IF

    IF (p==1) THEN
      xgl(1)=-0.57735
      xgl(2)=+0.57735
    END IF

    IF (p==2) THEN
      xgl(1)=-dsqrt(3.d0/5.d0)
      xgl(2)=0.0d0
      xgl(3)=dsqrt(3.d0/5.d0)
    END IF

    IF (p==3) THEN
      xgl(1)=-0.861136
      xgl(2)=-0.339981
      xgl(3)=0.339981
      xgl(4)=0.861136
    END IF

    IF (p==4) THEN
      xgl(1)=-0.90618
      xgl(2)=-0.538469
      xgl(3)=0.d0
      xgl(4)=0.538469
      xgl(5)=0.90618
    END IF

    IF (p==5) THEN
      xgl(1)=-0.9324695142031521
      xgl(2)=-0.6612093864662645
      xgl(3)=-0.2386191860831969
      xgl(4)=0.2386191860831969
      xgl(5)=0.6612093864662645
      xgl(6)=0.9324695142031521
```

```
    END IF

    IF (p==6) THEN
      xgl(1)=-0.9491079123427585
      xgl(2)=-0.7415311855993945
      xgl(3)=-0.4058451513773972
      xgl(4)=0.0000000000000000
      xgl(5)=0.4058451513773972
      xgl(6)=0.7415311855993945
      xgl(7)=0.9491079123427585
    END IF

    IF (p==7) THEN
      xgl(1)=-0.9602898564975363
      xgl(2)=-0.7966664774136267
      xgl(3)=-0.5255324099163290
      xgl(4)=-0.1834346424956498
      xgl(5)=0.1834346424956498
      xgl(6)=0.5255324099163290
      xgl(7)=0.7966664774136267
      xgl(8)=0.9602898564975363
    END IF

    IF (p==8) THEN
      xgl(1)=-0.9681602395076261
      xgl(2)=-0.8360311073266358
      xgl(3)=-0.6133714327005904
      xgl(4)=-0.3242534234038089
      xgl(5)=0.0000000000000000
      xgl(6)=0.3242534234038089
      xgl(7)=0.6133714327005904
      xgl(8)=0.8360311073266358
      xgl(9)=0.9681602395076261
    END IF

    IF (p==9) THEN
      xgl(1)=-0.9739065285171717
      xgl(2)=-0.8650633666889845
      xgl(3)=-0.6794095682990244
      xgl(4)=-0.4333953941292472
      xgl(5)=-0.1488743389816312
      xgl(6)=0.1488743389816312
      xgl(7)=0.4333953941292472
      xgl(8)=0.6794095682990244
      xgl(9)=0.8650633666889845
      xgl(10)=0.9739065285171717
    END IF

    END SUBROUTINE glnodes


!*********** Generate a 1D grid (x only) ***************
! Computing locations of element face
    SUBROUTINE grid1d
    IMPLICIT NONE
    INTEGER :: i,j, counter
    ! Grid spacing
    Dx = (b-a)/FLOAT(nels)
    ! Generate faces between elemenst
    x(1)=a
    j=2
    DO i =1, nfaces-2
      x(j) = a + (i)*Dx
      j=j+1
    END DO
    x(nfaces)=b

    counter=1
    WRITE (*,*) 'x values (faces):'
    DO i=1, nfaces
      WRITE (*,*) counter, x(i)
      counter=counter+1
    END DO
    END SUBROUTINE grid1d
```

```
!******************** Face to element mapping ************************
! For a given face, finding the element to its right (1) and left (2)
    SUBROUTINE face_to_element_map
    IMPLICIT NONE
    INTEGER :: i
    DO i=1, nfaces
      IF (i==1) THEN
        f2e(i,1)=i
        f2e(i,2)=nels
      ELSE IF (i==nfaces) THEN
        f2e(i,1)=1
        f2e(i,2)=i-1
      ELSE
        f2e(i,1)=i
        f2e(i,2)=i-1

      END IF
    END DO
    END SUBROUTINE face_to_element_map


!******************** Element to face mapping ************************
! For a given element, finding the face to its right (1) and left (2)
    SUBROUTINE element_to_face_map
    IMPLICIT NONE
    INTEGER :: i
    DO i=1, nels
      IF (i==1) THEN
        e2f(i,2)=nfaces
        e2f(i,1)=i+1
     ELSE IF(i==nels) THEN
        e2f(i,2)=i
        e2f(i,1)=1
      ELSE
        e2f(i,2)=i
        e2f(i,1)=i+1
      END IF
    END DO
    END SUBROUTINE element_to_face_map


!************** Provide intial condition ************************
    SUBROUTINE init1d
      IMPLICIT NONE
      INTEGER :: i, ppp1

      DO i=1, nels
        DO ppp1=1, pp1
        ! Finding x-space values of Gauss/internal points from xgl in ksi/standard space (-1,+1)
        ! [The values of xgl given in tables are in ksi space, so finding corr. x-space values to then find u and f]
        ! Equation 2.5 Vincent et al.(2011)
        x_internal(i,ppp1)=((1-xgl(ppp1))/2.0d0)*x(i)+((1+xgl(ppp1))/2.0d0)*x(i+1)
        u(i,ppp1)=ispeed*EXP(-20.0d0*x_internal(i,ppp1)**2) - (ispeed-1)*(SIN(pi*x_internal(i,ppp1)))
        ! Now we have provided initial conditions in the interior points (no values are there in the faces)
        END DO
      END DO

      u_ic=u

    IF (iwave==1) THEN ! Linear
      Dt = (cfl_ip*Dx)
    ELSE
      Dt = (cfl_ip*Dx)/MAXVAL(ABS(u)) ! Non-linear burger's
    END IF
    nptst = ABS(d-c)/Dt

    OPEN(unit = 0, file = 'initial_u.dat', status = 'replace')
      DO i = 1, nels
        DO ppp1=1, pp1
          WRITE(0, *)  x_internal(i,ppp1), u(i,ppp1)
        END DO
      END DO
    CLOSE(0)

    END SUBROUTINE init1d
```

```
!************** Discontinuous flux ************************
    SUBROUTINE fluxD
      IMPLICIT NONE
      INTEGER :: i, ppp1

      DO i=1, nels
        DO ppp1=1, pp1
        ! Compute Jacobian
        J=(x(i+1)-x(i))/2.0d0
          IF (iwave==1) THEN !Linear
            auu(i,ppp1)=1
          ELSE !Burger's
            auu(i,ppp1)=uold(i,ppp1) !Since u_l(i)=u_r(i), assigning au(i)=u_l(i)
          END IF
        ! Discontinuous flux and then normalizing it by the Jacobian
        ! Equation 2.8 Vincent et. al(2011)
        f(i,ppp1)=(iwave*auu(i,ppp1)*uold(i,ppp1) - (iwave-1.d0)*auu(i,ppp1)*(uold(i,ppp1)))/J
        END DO
      END DO

!******** Exact Solution *******
      DO i=1, nels
        DO ppp1=1, pp1
          IF (iwave==1) THEN !Linear
            a_ini(i,ppp1)=1
          ELSE !Burger's
            a_ini(i,ppp1)=u_ic(i,ppp1) !Since u_l(i)=u_r(i), assigning au(i)=u_l(i)
          END IF
        !u_iee(i,ppp1)=ispeed*EXP(-20.0d0*(x_internal(i,ppp1)-(a_ini(i,ppp1)*d))**2) - &
        !                          (ispeed-1)*(SIN(pi*(x_internal(i,ppp1)-(a_ini(i,ppp1)*d))))
        u_iee(i,ppp1)=ispeed*EXP(-20.0d0*(x_internal(i,ppp1)-(a_ini(i,ppp1)*d))**2) - &
                                 (ispeed-1)*(SIN(pi*(x_internal(i,ppp1)-(auu(i,ppp1)*d))))

        END DO
      END DO

    OPEN(unit = 8, file = 'exact_u.dat', status = 'replace')
      DO i = 1, nels
        DO ppp1=1, pp1
          WRITE(8, *)  x_internal(i,ppp1), u_iee(i,ppp1)
        END DO
      END DO
    CLOSE(8)

    OPEN(unit = 1, file = 'initial_f.dat', status = 'replace')
      DO i = 1, nels
        DO ppp1=1, pp1
          WRITE(1, *)  x_internal(i,ppp1), f(i,ppp1)
        END DO
      END DO
    CLOSE(1)

    END SUBROUTINE fluxD


!************** Compute Lagrange cardinal functions ************************
    SUBROUTINE lagrangePoly(xi,l)
      IMPLICIT NONE
      INTEGER :: k, j
      REAL(DP) :: term
      REAL(DP), INTENT(IN) :: xi
      REAL(DP), DIMENSION(1:pp1), INTENT(OUT) :: l
      DO k=1, pp1 ! Loop over each Lagrange polynomial from 1 to pp1
        term=1.0d0
        ! Product from 1 to pp1 (k.ne.j) to compute each Lagrange polynomial
        DO j=1, pp1
          IF (j.NE.k) THEN
            term=term*(xi-xgl(j))/(xgl(k)-xgl(j))
          END IF
        END DO
        l(k)=term
      END DO

      !WRITE(*, *) l
      !WRITE(*,*) '!!!Check-point: Sum of the above should=1'
```

```
    END SUBROUTINE lagrangePoly


!********* Extrapolating the internal values to the face using lagrange cardinal function **********
    SUBROUTINE extrapolate
      IMPLICIT NONE
      INTEGER :: i, j

      !Initializing u_face and f_face so as to take sum in the next sum
      DO i=1, nels
        DO j=1, 2
          u_face(i,j)=0.0d0
          f_face(i,j)=0.0d0
        END DO
      END DO

      DO i=1, nels
        DO j=1, pp1
          ! Right face of element i
          u_face(i,1)=u_face(i,1)+uold(i,j)*l_r(j)
          ! Left face of element i
          u_face(i,2)=u_face(i,2)+uold(i,j)*l_l(j)
          !Similarly for flux
          f_face(i,1)=f_face(i,1)+f(i,j)*l_r(j)
          ! Left face of element i
          f_face(i,2)=f_face(i,2)+f(i,j)*l_l(j)
        END DO
      END DO


    OPEN(unit = 2, file = 'f_left.dat', status = 'replace')
    DO i = 1, nfaces
!        WRITE(2, *) x(e2f(i,2)), f_face(i,2)
        WRITE(2, *) x(i), f_face(i,2)
    END DO
    CLOSE(2)
    OPEN(unit = 3, file = 'f_right.dat', status = 'replace')
    DO i = 1, nfaces
!        WRITE(2, *) x(e2f(i,2)), f_face(i,2)
        WRITE(3, *) x(i), f_face(i,1)
    END DO
    CLOSE(3)
    END SUBROUTINE extrapolate


!**************        Riemann problem            ************************
!************** Calculate Roe's interaction flux ************************

    SUBROUTINE interaction_flux
      IMPLICIT NONE
      INTEGER :: i

! Eq.2.19 Huynh(2007)
      DO i=1, nfaces !Looping through faces and NOT elements
        IF (u_face(f2e(i,2),1).NE.u_face(f2e(i,1),2)) THEN
          !u_l NE u_r (right face value of the left element NE to left face value of right element)
          !(f_r-f_l)/(u_r-u_l)
          au(i)=(f_face(f2e(i,1),2)-f_face(f2e(i,2),1))/(u_face(f2e(i,1),2)-u_face(f2e(i,2),1))
        ELSE !u_r=u_l
          IF (iwave==1) THEN !Linear
            au(i)=1.0d0
          ELSE !Burger's
            au(i)=u_face(f2e(i,2),1) !Since u_l(i)=u_r(i), assigning au(i)=u_l(i)
          END IF
        END IF
      END DO

! Eq.2.21 Huynh(2007)
      DO i=1, nfaces !Looping through faces and NOT elements
        f_interaction(i)=(0.5d0*(f_face(f2e(i,2),1)+f_face(f2e(i,1),2))-0.5d0*ABS(au(i))*(u_face(f2e(i,1),2)-u_face(f2e(i
      END DO

    OPEN(unit = 4, file = 'f_roe.dat', status = 'replace')
    DO i = 1, nfaces
!        WRITE(2, *) x(e2f(i,2)), f_face(i,2)
```

```
        WRITE(4, *) x(i), f_interaction(i)
    END DO
    CLOSE(4)
    END SUBROUTINE interaction_flux


!************** Calculate Lagrange Polynomial derivative ************************
! First calculates lagrange polynomial and then its derivative; hence four do loops
!Equation from https://math.stackexchange.com/questions/1105160/evaluate-derivative-of-lagrange-polynomials-at-construction
    SUBROUTINE lag_poly_der
      IMPLICIT NONE
      INTEGER :: k,m,l,j,i
      REAL(DP) :: lsum, term

      DO k=1, pp1
        DO m=1, pp1
          lsum=0.d0
          DO l=1, pp1
            term=1.0d0
            DO j=1, pp1
              IF ( j/=k .AND. j/=l ) THEN
                term=term*(xgl(m)-xgl(j))/(xgl(k)-xgl(j))
              END IF
            END DO
            IF (l/=k) THEN
              lsum=lsum+term/(xgl(k)-xgl(l))
            END IF
          END DO
        lpdm(m,k)=lsum
        END DO
      END DO

    OPEN(unit = 5, file = 'lag_poly_der.dat', status = 'replace')
    DO i = 1, pp1
        WRITE(5, *) lpdm(i,:)
    END DO
    CLOSE(5)

    END SUBROUTINE lag_poly_der


!************** Calculate correction function derivative ************************
! First hard-coding derivatives of p and p+1 legendre polynomials [derivatives of equations from slide (page 38, lec8-9) or
! Hard coding is not ideal as it restricts arbitrary order
! Ideally legendre polynomial should be contructed (page 51, lec.8-9) and its derivative should be computed (see two comment
    SUBROUTINE corr_fn_der
      IMPLICIT NONE
      INTEGER :: k,m,l,j,i
      REAL(DP) :: lsum, term

!!!Compute derivatives of Legendre polynomials
      !dlpdr(:)=(pp1*leg(:,p)-p*xgl(:)*leg(:,p))/(1.d0-xgl**2)
      !dlpp1dr(:)=(pp1*leg(:,p)-pp1*xgl(:)*leg(:,pp1))/(1.d0-xgl**2)

! Hard coding derivatives of Legendre polynomials
      IF (p==1) THEN
        dlpdr(:)=1.d0
        dlpp1dr(:)=3.0*xgl(:)
      ELSE IF (p==2) THEN
        dlpdr(:)= 3.0*xgl(:)
        dlpp1dr(:)=15.d0/2.d0*xgl(:)**2-1.5d0
      ELSE IF (p==3) THEN
        dlpdr(:)=15.d0/2.d0*xgl(:)**2-1.5d0
        dlpp1dr(:)=0.5d0*35.d0*xgl(:)**3-0.25d0*30.d0*xgl(:)
      ELSE IF (p==4) THEN
        dlpdr(:)=0.5d0*35.d0*xgl(:)**3-0.25d0*30.d0*xgl(:)
        dlpp1dr(:)=(1.0d0/8.0d0)*(63.d0*5.d0*xgl(:)**4-70.d0*3.d0*xgl(:)**2+15.d0)
      ELSE IF (p==5) THEN
        dlpdr(:)=(1.0d0/8.0d0)*(63.d0*5.d0*xgl(:)**4-70.d0*3.d0*xgl(:)**2+15.d0)
        dlpp1dr(:)=(1.d0/16.d0)*(231.d0*6.d0*xgl(:)**5-315.d0*4.d0*xgl(:)**3+105.d0*2.d0*xgl(:))
      ELSE IF (p==6) THEN
        dlpdr(:)=(1.d0/16.d0)*(231.d0*6.d0*xgl(:)**5-315.d0*4.d0*xgl(:)**3+105.d0*2.d0*xgl(:))
        dlpp1dr(:)=(1.d0/16.d0)*(429.d0*7.d0*xgl(:)**6-693.d0*5.d0*xgl(:)**4+315.d0*3.d0*xgl(:)**2-35.d0)
      ELSE IF (p==7) THEN
        dlpdr(:)=(1.d0/16.d0)*(429.d0*7.d0*xgl(:)**6-693.d0*5.d0*xgl(:)**4+315.d0*3.d0*xgl(:)**2-35.d0)
        dlpp1dr(:)=(1.d0/128.d0)*(6435.d0*8.d0*xgl(:)**7-12012.d0*6.d0*xgl(:)**5+6930.d0*4.d0*xgl(:)**3-1260.d0*2.d0*xgl(
```

```
      ELSE IF (p==8) THEN
        dlpdr(:)=(1.d0/128.d0)*(6435.d0*8.d0*xgl(:)**7-12012.d0*6.d0*xgl(:)**5+6930.d0*4.d0*xgl(:)**3-1260.d0*2.d0*xgl(:))
        dlpp1dr(:)=(1.d0/128.d0)*(12155.d0*9.d0*xgl(:)**8-25740.d0*7.d0*xgl(:)**6+18012.d0*5.d0*xgl(:)**4- &
                   4620.d0*3.d0*xgl(:)**2+315)
      ELSE IF (p==9) THEN
        dlpdr(:)=(1.d0/128.d0)*(12155.d0*9.d0*xgl(:)**8-25740.d0*7.d0*xgl(:)**6+18012.d0*5.d0*xgl(:)**4- &
                  4620.d0*3.d0*xgl(:)**2+315)
        dlpp1dr(:)=(1.d0/256.d0)/(46189.d0*10.d0*xgl(:)**9-109395.d0*8.d0*xgl(:)**7+90090.d0*6.d0*xgl(:)**5- &
                   30030.d0*4.d0*xgl(:)**3+3465.d0*2.d0*xgl(:))
      END IF

! Compute derivative of Radau polynomial
      dgl(:)=(-1.d0)**p*0.5d0*(dlpdr(:)-dlpp1dr(:))
      dgr(:)=0.5d0*(dlpdr(:)+dlpp1dr(:))

!      WRITE (*,*) 'dgl values:'
!      DO i=1, pp1
!        WRITE (*,*) dgl(i)
!      END DO
!      WRITE (*,*) 'dgr values:'
!      DO i=1, pp1
!        WRITE (*,*) dgr(i)
!      END DO
!      WRITE (*,*) 'Check-point: Sum of the terms in each row should=0'
    END SUBROUTINE corr_fn_der


!**************              Compute RHS               ************************
!************** Calculate divergence of continuous flux ************************
! Eq.3.9 Huynh(2007) or 2.17 Vincent et al.(2011) or pg.30 lec.8-9
    SUBROUTINE rhs
      IMPLICIT NONE
      INTEGER :: i, ppp1, k

      rhs1(:,:)=0.d0
! Vector-matrix multiplication (entire first term on RHS)
      DO i=1, nels
        DO ppp1=1, pp1
          DO k=1, pp1
            rhs1(i,ppp1)=rhs1(i,ppp1)+f(i,k)*lpdm(ppp1,k)
          END DO
        END DO
      END DO
! Computing RHS,i.e., divergence of flux
      DO i=1, nels
        DO ppp1=1, pp1
          df(i,ppp1)= -(rhs1(i,ppp1) + (f_interaction(e2f(i,2))-f_face(i,2))*dgl(ppp1) + &
                                       (f_interaction(e2f(i,1))-f_face(i,1))*dgr(ppp1))
        END DO
      END DO

    END SUBROUTINE rhs


!************ RK-3 *******************
  SUBROUTINE rk3
    USE types_vars
    !USE variables

    IMPLICIT NONE

    INTEGER :: i, j, k, ppp1

    uold = u

    !DO j=0, 1
    DO j=0, nptst

       !STEP 1

      CALL fluxD
      CALL extrapolate
      CALL interaction_flux
      CALL lag_poly_der
      CALL corr_fn_der
      CALL rhs
```

```
      k1=Dt*df
      uold = u + k1

       !STEP 2
    CALL fluxD
    CALL extrapolate
    CALL interaction_flux
    CALL lag_poly_der
    CALL corr_fn_der
    CALL rhs

       k2=(1.d0/4.d0)*Dt*df
       uold=((3.d0/4.d0)*u)+((1.d0/4.d0)*uold)+k2

       !STEP 3
    CALL fluxD
    CALL extrapolate
    CALL interaction_flux
    CALL lag_poly_der
    CALL corr_fn_der
    CALL rhs

       k3=(2.d0/3.d0)*Dt*df
       uold=((1.d0/3.d0)*u)+((2.d0/3.d0)*uold)+k3


      u=uold
      t=t+Dt
      WRITE(*,*) 'Total time(s)=',t

    END DO

  !  OPEN(unit = 7, file = 'sol_case1_h40p2.dat', status = 'replace')
    OPEN(unit = 7, file = 'sol.dat', status = 'replace')
      DO i = 1, nels
        DO ppp1=1, pp1
          WRITE(7, *)  x_internal(i,ppp1), u(i,ppp1)
        END DO
      END DO
    CLOSE(7)

    WRITE(*,*) 'Time-step size=', Dt
    WRITE(*,*) 'CFL=', cfl_ip

    IF (iwave==1) THEN !For linear Gaussian profile using IC for comparison
      u_dd=ABS(u_ic-u)
    ELSE
      u_dd=ABS(u_iee-u) !For Burger's using exact solution for comparison
    END IF

    l1_rk3=SUM(u_dd)/SIZE(u_dd)
    l2_rk3=SQRT(SUM(u_dd**2))/SIZE(u_dd)
    l_infty_rk3=MAXVAL(u_dd)/SIZE(u_dd)
    WRITE(*,*) 'No. of elements(nels)=',nels
    WRITE(*,*) 'Grid size(h)=', Dx
    WRITE(*,*) 'Order of polynomial(p)=',p,':'
    WRITE(*,*) 'L_infty_NORM=', l_infty_rk3
    WRITE(*,*) 'L1_NORM=', l1_rk3
    WRITE(*,*) 'L2_NORM=', l2_rk3

  END SUBROUTINE rk3


!*********** Print out RHS file*******************
    SUBROUTINE output_rhs
      IMPLICIT NONE
      INTEGER :: i, ppp1


    OPEN(unit = 6, file = 'rhs.dat', status = 'replace')
      DO i = 1, nels
        DO ppp1=1, pp1
          WRITE(6, *)  x_internal(i,ppp1), df(i,ppp1)
        END DO
```

```
    END DO
  CLOSE(6)

  END SUBROUTINE output_rhs


END MODULE subroutines



!************************************************************************
!************************************************************************
!************************************************************************
!************************************************************************
! MAIN PROGRAM
PROGRAM FRM

  USE types_vars
  USE variables
  USE subroutines

  CALL inputs
  CALL memalloc
  CALL glnodes
  CALL grid1d
  CALL face_to_element_map
  CALL element_to_face_map
  CALL init1d
  CALL lagrangePoly(-1.0d0,l_l) ! l_l is array (1:pp1) of scalar values of lagrange basis functions at face -1
  CALL lagrangePoly(1.0d0,l_r) ! l_r is array (1:pp1) of scalar values of lagrange basis functions at face +1
  CALL rk3
  CALL output_rhs
  CALL dealloc

END PROGRAM FRM
```