UNIVERSITY OF TENNESSEE AT CHATTANOOGA
Department of Computational Engineering

Validation of a three-dimensional unstructured flow solver

Flow Solver Course provided by Prof. K. Screenavis

prepared by

Arash Ghasemi
May 25, 2012

Table of Contents

List of Figures

Abstract

In this report, we implement and validate a three-dimensional algorithm for solving inviscid Euler equations on general geometries represented by a single block for a fair regime of fluid flow. To this end, we first use a set of predefined routines to read the grid topology. The input file is assumed to be in UGRID format. The grid topology is defined using the number of nodes, number of different elements in the grid, coordinates of all nodes, definitions for various elements including triangle and quadlateral for boundaries, and tetrahederal, pyramid, prism and hex for the interior regions. Prior to writing grid maps, we implement different element-based maps which locally specify the relation between face and node, relation between node and nodes, and the relation between edge and node for a particular element. Using these local maps, we proceed to implement various gird maps to access the grid point/edge of interest in the flow-solver section. These maps include, element sourrounding a point, points surrounding a particular point, edges sorrounding point, and element sourrounding an element. In the second part of of the work, we transfer the cell-centered grid to the equal node centered representation which is more suitable for finite volume approach. In doing so, we define our geometrical data structure based on the edges in the interior of the domain and ghost edges on the boundaries. For each triangle boundary element, we define three ghost edges while for each quad, we will have four. The volume of the new node-centered cell together with conservative variables are stored in node-based array. All maps and transformed node-centered grid are validated using a set of predefined meshes including a 13 node generic grid, an all-hexahederal and all-tetrahederal meshes, a course and fine ramp mesh for supersonic evaluations and a heavy visoucs mesh for NACA0012 airfoil. In the third part of the work, we implement the flow-solver. The conservation-laws are discretized using finite-volume approach [1] and a generic equation for temporal update is obtained. Flux vectors are using Roe approach. Roe averages are evaluated at the center of

[1] integration in space + Divergence theorem)

edges providing a Jacobian-based first-order approximation of flow field in space. For temporal discretization, both first-order Euler-explicit and fourth-order classic Runge-Kutta are implemented. A fifth-order Runge-Kutta scheme from Jameson-Baker is also implemented in the code. To validate the current algorithm, a supersonic ramp flow is solved on both coarse and refined meshes. Convergence study is performed on both coarse and refined meshes. The shock angle and various parameter before and after shock are compared with the one-dimensional oblique shock theory. The method is extended to second-order in space by using least square method for gradient construction. In addition, viscous terms are added by averaging the gradients over an edge. The final solver is validated for several benchmark problem including nonlinear acoustics of standing waves inside a pipe, vortex propagation and laminar boundary layer over flat plates. A comparison of the results with experimental data and analytical results for boundary layers proves the robustness of current solver.

1.0 Finding maps for the input cell-based grid

In this section we implement element-based maps which locally specify the relation between face and node, relation between node and nodes, and the relation between edge and node for a particular element. Using these local maps, we proceed to implement various gird maps to access the grid point/edge of interest in the flow-solver section. These maps include, element sourrounding a point, points surrounding a particular point, edges sorrounding point, and element sourrounding an element.

1.1 Before finding maps

Before doing anything we need to write a set of perliminary routines which are used in map algorithms. These include global to local and local to global transformation functions for all elements. In order to get a global element number (gelem) given a local element number (ielem) and element type (etype), we use the following code.

Listing 1: function for transforming local element number to a global number based on the order of elements

```
int Local_Elem_To_Global_Elem(int ielem,
                              int nelem[],
                              int etype)
{
    //Declaring index variables
    int e;

    //Declaring output and initialization
    int gelem = ielem;

    for(e = Triangle; e < etype; e++)
        gelem += nelem[e];

    return gelem;

}
```

Also, we need the inverse of local-to-global function given in listing 1 for the given element. The inverse can be implemented as the following,

Listing 2: function for obtaining the local information of each element based on the global number of the element

```
void Global_Elem_To_Local_Elem(int *ielem,
                               int nelem[],
                               int gelem,
                               int *etype)
{

    //Declaring index variables
    int e;

    //Initializing the local element index
    *ielem = gelem;
    for( e = Triangle; e <= Hex; e++)
        if((*ielem) <= nelem[e]) //if the element fits within a type
        {
            *etype = e; //everything is done!
            break;
        }
        else //otherwise we should subtract the previous element indices
            (*ielem) -= nelem[e];

}
```

1.2 The first map: element surrounding a node

To find element surrounding a node we use Compressed Row Storage (CRS) in the code. A linked-list arrays (hash-table) is also implemented. The CRS is efficiently stores the maps with minimum allocation. The lisked-list approach is faster because we don't need the pre-calculate the size of required memory. We use the following algorithm to store element surrounding a point in a CRS array.

Listing 3: general algorithm for obtaining elements surrounding a point (used from Ref.([1]))

```
Input: Number of nodes (nnodes)
Input: Number of elements (nelem[])
Input: Element connectivity (*c2n[])
Output: An index array nnodes + 2 long (nesp[])
Output: List of elements surrounding a node (esp[])
```

```
(1) Initialize nesp to 0.
(2) Fill in the entries for nesp
Loop over element types (Triangle <= etype <= Hex)
Loop over elements of type etype (1 <= ielem <= nelem[etype])
Loop over nodes of the element (0 <= inode <= mnode)
node = c2n[etype][mnode*ielem + inode]
nesp[node + 1]++
End Loop over inode
End Loop over ielem
End Loop over etype
(3) Generate the offsets for the CRS array
nesp[i] += nesp[i - 1] (2 <= i <= nnodes + 1]
(4) Allocate storage for esp
esp will be nesp[nnodes + 1] long
(5) Repeat Step (2) above, but store data
Loop over element types (Triangle <= etype <= Hex)
Loop over elements of type etype (1 <= ielem <= nelem[etype])
Loop over nodes of the element (0 <= inode <= mnode)
node = c2n[etype][mnode*ielem + inode]
indx = nesp[node]
esp[indx] = LocalElemToGlobalElem(ielem, nelem, etype)
nesp[node]++
End Loop over inode
End Loop over ielem
End Loop over etype
(6) The CRS indices are shifted by 1 after Step (5), so shift them back
Loop over nodes (nnodes + 1 >= i > 1)
nesp[i] = nesp[i - 1]
End Loop over nodes
In order to access the list of elements surrounding a point, do the following:
indx1 = nesp[i + 0]
indx2 = nesp[i + 1]
for (indx = indx1; indx < indx2; indx++) ielem = esp[indx];
```

Here is the final source code using CRS implementation.

Listing 4: CRS implementation of elements surrounding a point map

```
int Gen_Map_Elem__Sur_Point(int *nnodes,
                            int nelem[],
                            int*c2n[],
                            int **nesp,
                            int **esp
                            )
{
    //Declaring index variables
    int ii, ielem, inode, mnode, node, indx;
    ElemType etype;

    //Allocating memory for nesp
    *nesp = (int *) calloc((*nnodes) + 2, sizeof(int));

    //1 - Initializing nesp to ZERO
    for(ii = 0; ii < (*nnodes) + 2; ii++)
```

```
        *(*nesp + ii) = 0;

    //2- Filling in the entries for nesp
    for (etype = Triangle; etype <= Hex; etype++)
        for (ielem = 1; ielem <= nelem[etype]; ielem++)
        {
            mnode = ElemTypeToMnode(etype);
            for(inode = 0; inode < mnode; inode++)
            {
                node = c2n[etype][mnode*ielem + inode];
                (*nesp)[node + 1]++;
            }

        }
    //3- Generate the offset for the CRS array

    for(ii = 2; ii <= ((*nnodes) + 1); ii++)
        (*nesp)[ii] += (*nesp)[ii - 1];

    //4- Allocating memory for esp
    *esp = (int *) calloc((*nesp)[(*nnodes) + 1], sizeof(int));

    //5- Reapeating (2) , but storing the data
    for (etype = Triangle; etype <= Hex; etype++)
        for (ielem = 1; ielem <= nelem[etype]; ielem++)
        {
            mnode = ElemTypeToMnode(etype);
            for(inode = 0; inode < mnode; inode++)
            {
                node = c2n[etype][mnode*ielem + inode];
                indx = (*nesp)[node];
                (*esp)[indx] = Local_Elem_To_Global_Elem(ielem,nelem,etype);
                (*nesp)[node]++;
            }

        }

    //6- The CRS indices are shifted by 1 after Step (5)
    for( ii = ((*nnodes) + 1); ii >= 1; ii--)
        (*nesp)[ii] = (*nesp)[ii-1];


    //Operations completed successfully!
    return 0;

}
```

1.2.1 Results

The results are presented in table (1).

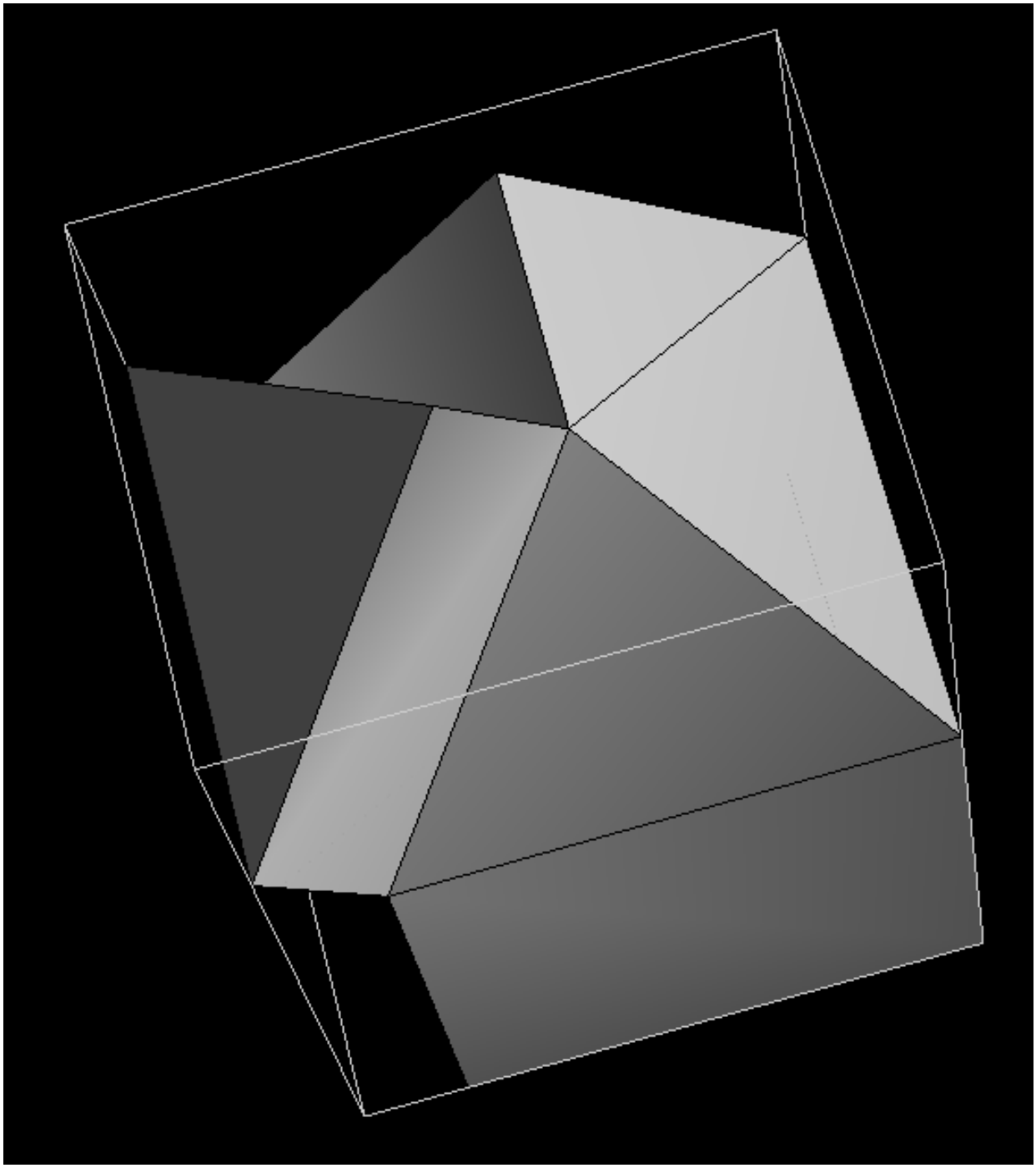


Figure 1: The very famous 13 points grid with the complexity of billion nodes mesh.

Grid	Elements surrounding point
13-node.ugrid	6,13,14,17
hex.ugrid	5,9,69,73,98,99
tet.ugrid	35,37,123,125,429,691
ramp-coarse.ugrid	6,7,527,2541,2561,2986,2987,3507
naca0012.ugrid	66,6839,25300,25340,39970,46743,66296,66297

Table 1: The result of element surrounding a point map for point 10 for different grids.

1.3 Nodes surrounding a point

Using the element surrounding a point we are able now to create the nodes surrounding a node map. This map is the most important map in the realm of node-centered finite volume schemes since all edges are constructed based on this map. To acheive the most efficient implementation, we use the general algorithm described in Ref.(**??**). Here we write a CRS version and a linked-list array version. To make this report standalone, we bring the original algorithm in the following listing.

Listing 5: General algorithm for obtaining nodes surrounding a point (used from Ref.([1]))

```
Input: Number of nodes (nnodes)
Input: Number of elements (nelem[])
Input: Element connectivity (*c2n[])
Input: List of elements surrounding point index (nesp[])
Input: List of elements surrounding a node (esp[])
Output: An index array nnodes + 2 long (npsp[])
Output: List of points surrounding a point (psp[])
(1) Build the list of elements surrounding a point
(2) For each element type, create a lookup table that has the list of nodes surrounding
a given node. A surrounding node is one that is directly connected to the node of
interest via an edge. For example, in a pyramid, we have 5 nodes and 8 edges.
Referring to the pyramid figure, we can see that the nodes surrounding node 1
are 2, 4 and 5. Similarly for nodes 2, 3 and 4. However, for node 5, we have the
surrounding nodes as 1, 2, 3 and 4. Therefore, you need to allow for a variable
number of surrounding nodes. This is best implemented using a CRS format
discussed in the element surrounding a point map algorithm. The same is true for
the points surrounding a point map in general. You do not know, a priori, how
many nodes you have surrounding a given node, so it is best to use a CRS format
to store the map.
(3) Loop over nodes (1 <= inode <= nnodes)
Loop over elements surrounding node (gelem)
Determine the element type (etype) and local element number (ielem)
Use a GlobalElemToLocalElem(gelem, nelem, &etype, &ielem)
function to get ielem and etype given gelem (global element number)
In the current element, find the location of inode (call it lnode)
Create a list of the nodes of ielem that surround lnode
```

```
Use results from Step (2) to help above
End loop over elements surrounding node
Eliminate duplicates from the list that was created above and create a list of
unique nodes.
Store the list using a CRS format as each node can have a variable number of
nodes surrounding it.
End loop over nodes
(4) Perform sanity checks: If node1 is in the surrounding node list of node2, then
node2 better be in the surrounding list of node1.
```

The following contains code listing for CRS version of nodes surrounding a point.

Listing 6: CRS implementation for nodes surrounding a point

```
int Gen_Map_Point__Sur_Point_LookUp(int *nnodes,
                            int nelem[],
                            int*c2n[],
                            int **nesp,
                            int **esp,
                            int your_point, int **surr_points, struct int_vec **enct)
{
    //Defining local variables
    int indx,indx1,indx2;
    int mnode,inode,node;
    int gelem, ielem , etype;
    int point_index;

    //For the given point, find all surrounding elements
    indx1 = (*nesp)[your_point + 0];
    indx2 = (*nesp)[your_point + 1];
    for(indx = indx1; indx < indx2; indx++)
    {
        gelem = (*esp)[indx]; //get the golobal index of the neighbur
        //find the local parameters of that element
        Global_Elem_To_Local_Elem(&ielem,nelem,gelem,&etype);
        mnode = ElemTypeToMnode(etype); //find the number of nodes in element
        //find the index of given node (one-based index) in that
        //element so that we can recognize neighbouring points
        for(inode = 0; inode < mnode; inode++)
            if( your_point == c2n[etype][mnode*ielem + inode])
                point_index = inode+1;
        //detect appropriate surrounding nodes
        for(inode = 1; inode < (*enct)->nexts[etype].nexts[point_index].length; inode++)
        {
            //find the local number of surrounding points in that element
            node = (*enct)->nexts[etype].nexts[point_index].int_data[inode];
            //convert the the local node of line above to the corresponding
            //global number of node.
            node = c2n[etype][mnode*ielem + node-1];
            //put a tick in the position where surrounding node appeares
            (*surr_points)[node]++;
        }
    }
```

```
41      return 0; //completed successfully
}

int Gen_Map_Point__Sur_Point_CRS(int *nnodes,
                                 int nelem[],
                                 int *c2n[],
                                 int **nesp,
                                 int **esp,
                                 int **npsp,
                                 int **psp)
{
    //declaring local variables
    int innodes = 0, j = 0, ngb_size = 0, CRS_i = 0;
    //decleration of element-node map holder pointer
    struct int_vec *enct;

    Create_Element_Node_Conn_Table(&enct);

    //allocating memory for CRS array indexer
    *npsp = (int *)calloc(*nnodes+2, sizeof(int));

    //Initializing CRS indexer
    for(CRS_i = 0; CRS_i < (*nnodes+2); CRS_i++)
        (*npsp)[CRS_i] = 0;

    //Allocating surr_points array with the same size of *nnodes + 1
    int *surr_points = (int *)calloc(*nnodes + 1,sizeof(int));
    //Initializing surr_points to zero
    for(j = 1; j <= *nnodes; j++)
        *(surr_points+j) = 0;

    //(*) Proceeding to determine the number surrounding points of all points
    for(innodes = 1; innodes <=  *nnodes; innodes++)
    {

        //Get the neighbour points for innodes
        Gen_Map_Point__Sur_Point_LookUp(nnodes,nelem,c2n,nesp,esp,innodes,&surr_points,&enct);

        //Calculate the number of neighbouring points
        for(j = 1; j <= *nnodes; j++)
            if(surr_points[j] != 0)
                ngb_size++;

        (*npsp)[innodes+1] = ngb_size; //storing the number of surr points
        //for node "innodes" in the CRS indexer

        //resetting "ngb_size" for the next array
        ngb_size = 0;
        //resetting surr_points
        for(j = 1; j <= *nnodes; j++)
            surr_points[j] = 0;


    }
    //Generating the offset for CRS array
    for(CRS_i = 2; CRS_i <= (*nnodes+1); CRS_i++)
```

```
97          (*npsp)[CRS_i] += (*npsp)[CRS_i-1];
        //Allocating CRS array for psp based on indexer
99      *psp = (int *)calloc((*npsp)[*nnodes+1], sizeof(int));
        //Reapiting the operation in (*), but storing the points at this stage
101     for(innodes = 1; innodes <= *nnodes; innodes++)
        {
103         //Get the neighbour points for innodes
            Gen_Map_Point__Sur_Point_LookUp(nnodes,nelem,c2n,nesp,esp,innodes,&surr_points,&enct);
105         //Now, proceed to store neighbour points
            for(j = 1; j <= *nnodes; j++)
107             if(surr_points[j] != 0)
                {
109                 (*psp)[(*npsp)[innodes]] = j;
                    (*npsp)[innodes]++;
111             }
            //resetting surr_points
113         for(j = 1; j <= *nnodes; j++)
                surr_points[j] = 0;
115     }
        //Shifting CRS indices by 1
117     for(CRS_i = (*nnodes+1); CRS_i >= 1; CRS_i--)
            (*npsp)[CRS_i] = (*npsp)[CRS_i-1];
119
/*      //Printing the output
121     for(CRS_i = 0; CRS_i < (*npsp)[*nnodes+1]; CRS_i++)
            printf("%d\n\r",(*psp)[CRS_i]);
123 */

125     //Performing clean-up for arrays allocated inside the function
        free(surr_points);
127     free(enct);

129 return 0; //return successfully
    }
```

The linked-list array version code is presented below.

Listing 7: Linked-list array implementation for nodes surrounding a point

```
    int Gen_Map_Point__Sur_Point(int *nnodes,
2                                int nelem[],
                                 int*c2n[],
4                                int **nesp,
                                 int **esp,
6                                struct int_vec **psp)
    {
8       //declaring local variables
        int innodes = 0, j = 0, ngb_size = 0, k = 0;
10
        //decleration of element-node map holder pointer
12      struct int_vec *enct;

14      Create_Element_Node_Conn_Table(&enct);
```

```
16      //initializing the ROOT of output vector containing neighbours for all
        //points in the domain.
18      *psp = (struct int_vec *)calloc(1,sizeof(struct int_vec));
        (*psp)->length = *nnodes;
20      (*psp)->nexts = (struct int_vec *)calloc((*nnodes+1),sizeof(struct int_vec));

22      //Allocating surr_points array with the same size of *nnodes + 1
        int *surr_points = (int *)calloc(*nnodes + 1,sizeof(int));
24      //Initializing surr_points to zero
        for(j = 1; j <= *nnodes; j++)
26          *(surr_points+j) = 0;

28      //Proceeding to determine surrounding points of all points
        for(innodes = 1; innodes <=  *nnodes; innodes++)
30      {
            //Get the neighbour points for innodes
32          Gen_Map_Point__Sur_Point_LookUp(nnodes,nelem,c2n,nesp,esp,innodes,&surr_points,&enct);

34          //Calculate the number of neighbouring points
            for(j = 1; j <= *nnodes; j++)
36              if(surr_points[j] != 0)
                    ngb_size++;
38
            //Creating appropriate array with size "ngb_size" for storing the
40          //neighburs of "innodes"
            (*psp)->nexts[innodes].int_data = (int *)calloc(ngb_size,sizeof(int));
42          (*psp)->nexts[innodes].length = ngb_size;

44          //Now, proceed to store neighbour points
            for(j = 1; j <= *nnodes; j++)
46              if(surr_points[j] != 0)
                {
48                  (*psp)->nexts[innodes].int_data[k] = j;
                    k++;
50              }
            //resetting "k" for the next array
52          k = 0;
            //resetting "ngb_size" for the next array
54          ngb_size = 0;
            //resetting surr_points
56          for(j = 1; j <= *nnodes; j++)
                surr_points[j] = 0;
58
        } //done for all points!
60
        //Performing clean-up for arrays allocated inside the function
62      free(surr_points);
        free(enct);
64
    return 0; //return successfully
66  }
```

Note that in listings (7) and (6) we used the local map Create-Element-Node-Conn-Table which is a big linked-list table that exactly defines the connectivity map for each node in a particular element. For example for the local node 2 in a triangle element, it returns local indices 1 and 3.

1.3.1 Results

The results for CRS and linked-list array are presented in table (2).

Grid	Nodes surrounding a point
13-node.ugrid	11, 12, 13
hex.ugrid	9, 11, 46, 64
tet.ugrid	9, 11, 54, 84
ramp-coarse.ugrid	9, 11, 231, 232, 901
naca0012.ugrid	389, 792, 857, 1598, 14003

Table 2: The result of nodes surrounding a point map for point 10 for different grids.

1.4 Edge to node map

An edge connects two nodes. We always create edges such that they connect a lower numbered node to a higher numbered node. The key is to be consistent and use the same convention everywhere. The steps for creating the edge to node map are described in listing (8).

Listing 8: Algorithm for generating interior edges based on the point surrounding a point map

```
Input: An index array nnodes + 2 long (npsp[])
Input: List of points surrounding a point (psp[])
Output: Number of edges (nedge)
Output: Edge-to-node map (e2n[])
(1) Initialize nedge to zero
(2) Loop over nodes (1 <= inode <= nnodes)
Loop over nodes surrounding inode (jnode)
if (jnode > inode)
```

```
nedge++;
e2n[2*nedge + 0] = inode;
e2n[2*nedge + 1] = jnode;
end if
End loop over jnode
End loop over inode
```

The algorithm (8) is implemented in a single function described below.

Listing 9: The final implementation of edges to node map

```
1  int Gen_Edg_Point(struct int_vec **psp,int *nedge, int **e2n)
   {
3      //initializing local variables
       int inode = 0, jnode =0; //for indexing domain nodes
5      int estim = *nedge = 0; //for measuring the number of edges

7      //Estimating the number of edges
       for(inode = 1; inode <= (*psp)->length; inode++)
9          for(jnode = 0; jnode < (*psp)->nexts[inode].length; jnode++)
               if((*psp)->nexts[inode].int_data[jnode] > inode)
11                 estim++;
       //Allocating the edge array
13     *e2n = (int *) calloc(2*estim+2, sizeof(int));

15     //Filling the edge array
       for(inode = 1; inode <= (*psp)->length; inode++)
17         for(jnode = 0; jnode < (*psp)->nexts[inode].length; jnode++)
               if((*psp)->nexts[inode].int_data[jnode] > inode)
19             {
                   (*nedge)++;
21                 //Putting the edge in ascending format
                   (*e2n)[2*(*nedge)+0] = inode;
23                 (*e2n)[2*(*nedge)+1] = (*psp)->nexts[inode].int_data[jnode];
               }
25   //return successfully
     return 0;
27 }
```

1.4.1 Results

The code listing (9) is run for the set of desired grids. The final results for 13 nodes grid are presented below.

Edge number	Nodes forming that edge
1	1-2
2	1-4
3	1-8
4	2-3
5	2-5
6	2-9
7	2-11
8	2-13
9	3-4
10	3-6
11	3-12
12	3-13
13	4-7
14	5-6
15	5-8
16	5-9
17	5-13
18	6-7
19	6-13
20	7-8
21	9-13
22	10-11
23	10-12
24	10-13

Table 3: Edges number versus edges nodes for 13 nodes grid.

1.5 The fourth map: Elements surrounding an element

To obtain this map, we use an efficient algorithm. We first allocate a zero array with the size of nelem+1. Then we loop over elements and for each point in each element we find the surrounding elements using previous maps. We note that in the allocated array we increase the index of the element that surrounds that point. After this. we find the indices of the elements that are greater than or equal to 3. These are the surrounding element for a given element. The result of the algorithm is brought in Table (4).

Element Number	Surr. Element
1	15
2	15
3	15
4	16
5	16
6	17
7	18
8	18
9	18
10	18
11	18
12	17
13	17
14	17
15	1,2,3,16
16	4,5,15,17,18
17	6,12,13,14,16
18	7,8,9,10,11,16

Table 4: Validation of Element Surrounding Element for 13-node grid.

1.6 Cell-Centered to Node-Centered Transformation

Following Ref.([1]), we implemented the dual computing scheme. The result of area summation and volume summation are presented in Table (5). All areas are zero while the algorithm leads to correct value for the volume.

Grid Name	Ax	Ay	Az	Vol
13-node	1.387779e-17	0.000000e+00	-2.775558e-17	1.308333e+00
Hex	0.000000e+00	0.000000e+00	0.000000e+00	1.000000e+00
Tet	1.406861e-15	5.186823e-16	5.186823e-16	1.000000e+00
Ramp Coarse	3.483498e-14	2.307009e-14	-9.635626e-13	6.303005e+01
Ramp Refined	3.517152e-16	1.747734e-16	-5.758495e-15	9.665064e-02
Naca0012	1.554312e-15	-1.943151e-14	2.410048e-12	1.568258e+03

Table 5: Validation of Cell centered to Node centered transformation.

1.7 Renumbering

In this section, we use Cuthill-KcKee's algorithm to renumber the point surrounding matrix. The results are shown in figs.(2) and figs.(3). As shown in fig.(2), for the 13 node grid, there is no discernible difference between two grids. However for hex grid, the difference is clearly visible. (compare fig.(2) bottom left and right). We did the same thing for ramp coarse and tet grid and the results are shown in fig.(3) top and bottom respectively.

1.8 One-dimensional shubin nozzle

We have a set of one-dimensional Euler equations that is usually called shubin nozzle problem. In Ref.[1], equation $e = C_v T$ is the definition of the internal energy for a calorically perfect gas. Also $h = C_p T$ is the same approach for enthalpy. The total energy e_t is defined as the sum of the internal energy e plus the kinetic energy of the flow which is expressed in the term of velocity of the flow as $1/2(u^2 + v^2 + w^2)$. Since this is a one-dimensional problem, we can write $e_t = e + 1/2u^2$ $(v = w = 0)$. An also if we add the work done by pressure to the internal energy definition, we will get the definition for the enthalpy as $h_t = h + 1/2u^2$. In linear Acoustics, it can be shown that for an isentropic wave (sound wave) propagating in a lossless fluid, the pressure change due to density change is always constant and depends only on the ration of the specific heats $\gamma = C_p/C_v$ and the velocity of that propagating wave. This relation can be readily written as $c = \sqrt{(\gamma p/\rho)}$.

We can easily find the relation between conservative and primitive variables using the following equations.

Assuming $Q1 = \rho$, $Q2 = Q1u$, $Q3 = Q1e_t$, we can rewrite the equations in the form of conservative variables so that we can easily find the Jacobians. First we find the primitive variables and other variables of interest in the form all conservative variables. This is done

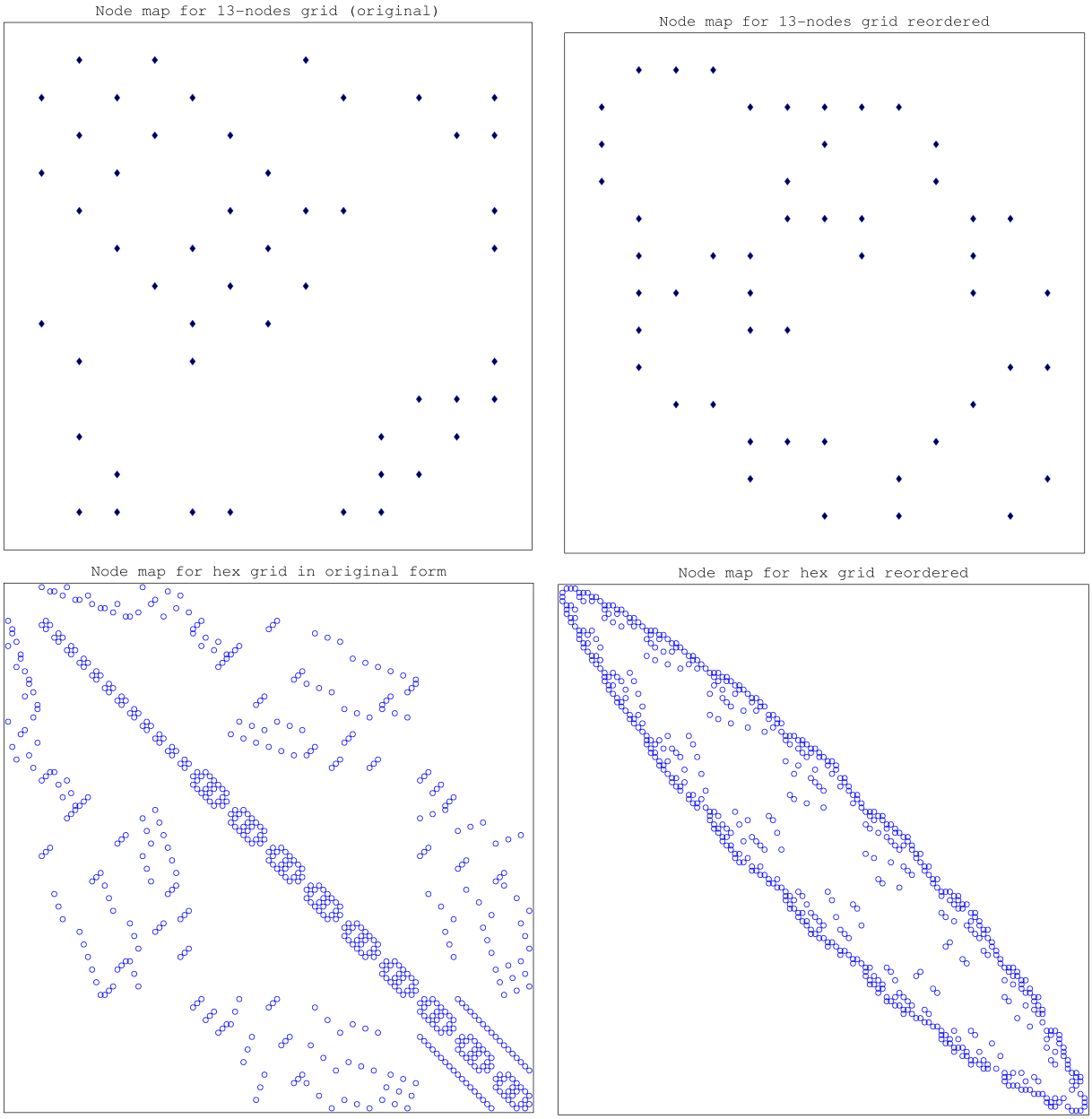


Figure 2: The original versus reordered grids.

as the following, For energy, we have,

$$e = \frac{Q3}{Q1} - 1/2\,\frac{Q2^2}{Q1^2} \tag{1}$$

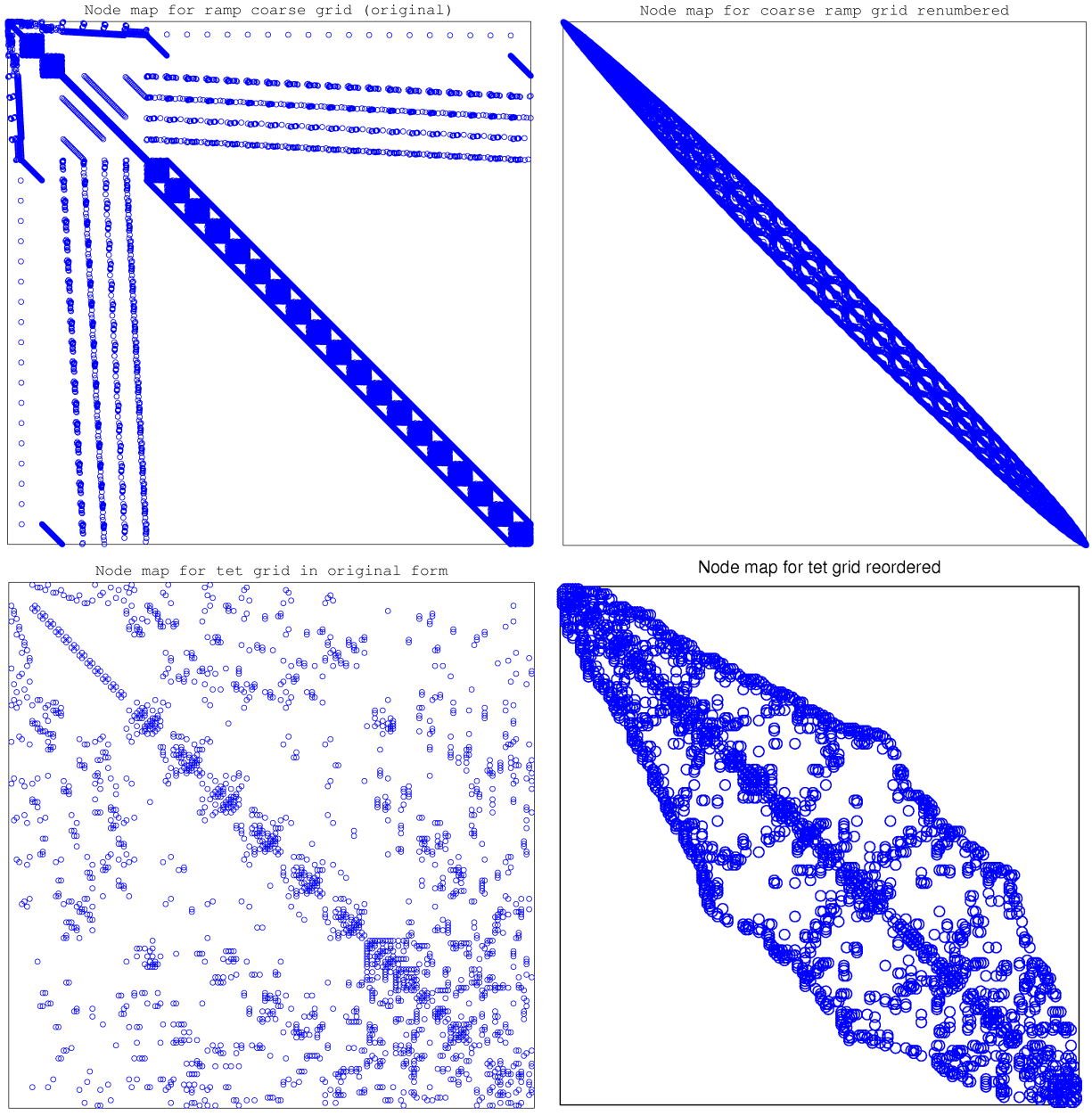


Figure 3: The original versus reordered grids.

For the pressure we can write,

$$p = Q1 \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (\gamma - 1) \tag{2}$$

Also for sound speed the substitution leads to,

$$c = \sqrt{-\gamma \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (-\gamma + 1)} \tag{3}$$

An for enthalpy and total enthalpy we obtain, for enthalpy,

$$h = -\gamma \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (-\gamma + 1) (\gamma - 1)^{-1} \tag{4}$$

$$h_t = -\gamma \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (-\gamma + 1) (\gamma - 1)^{-1} + 1/2 \frac{Q2^2}{Q1^2} \tag{5}$$

Using equations (**??**) the one-dimensional flux vector $f = [\rho, \rho u^2 + p, \rho h_t u]^T$ can be rewritten as,

$$f = \begin{bmatrix} Q2 \\ \frac{Q2^2}{Q1} + Q1 \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (\gamma - 1) \\ \left(-\gamma \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (-\gamma + 1) (\gamma - 1)^{-1} + 1/2 \frac{Q2^2}{Q1^2} \right) Q2 \end{bmatrix} \tag{6}$$

Then we can find the Jacobians as described below. First for $A = \frac{\partial f}{\partial Q}$ we derive,

$$A[col = 1] = \begin{bmatrix} 0 \\ -\frac{Q2^2}{Q1^2} + \left(\frac{Q3}{Q1} - 1/2 \frac{Q2^2}{Q1^2} \right) (\gamma - 1) + Q1 \left(-\frac{Q3}{Q1^2} + \frac{Q2^2}{Q1^3} \right) (\gamma - 1) \\ \left(-\gamma \left(-\frac{Q3}{Q1^2} + \frac{Q2^2}{Q1^3} \right) (-\gamma + 1) (\gamma - 1)^{-1} - \frac{Q2^2}{Q1^3} \right) Q2 \end{bmatrix} \tag{7}$$

$$A[col=2]=\begin{bmatrix} 1 \\ 2\,\frac{Q2}{Q1}-\frac{Q2\,(\gamma-1)}{Q1} \\ \left(\frac{Q2\,\gamma\,(-\gamma+1)}{Q1^2(\gamma-1)}+\frac{Q2}{Q1^2}\right)Q2-\gamma\,\left(\frac{Q3}{Q1}-1/2\,\frac{Q2^2}{Q1^2}\right)(-\gamma+1)\,(\gamma-1)^{-1}+1/2\,\frac{Q2^2}{Q1^2} \end{bmatrix} \tag{8}$$

$$A[col=3]=\begin{bmatrix} 0 \\ \gamma-1 \\ -\frac{Q2\,\gamma\,(-\gamma+1)}{Q1\,(\gamma-1)} \end{bmatrix} \tag{9}$$

Which has three eigen-values as

$$\begin{aligned} \lambda_{1,2} &= Q1\,\left(\frac{Q3}{Q1}-1/2\,\frac{Q2^2}{Q1^2}\right)(\gamma-1)\,, \\ \lambda_3 &= Q2/Q1, \end{aligned} \tag{10}$$

As shown in eq.(10) the eigen-values of matrix A are long and they don't have a major physical meaning. However, using conservative to primitive transformation we can find the eigen values with physical meaning (wave speeds) which are essential to impose characteristic based boundary conditions. If we differentiate primitive variables with respect to conservative ones, we will obtain the transformation matrix as below,

$$M=\frac{\partial[q]}{\partial[Q]}=\begin{bmatrix} 1 & 0 & 0 \\ \frac{Q2}{Q1} & Q1 & 0 \\ 1/2\,\frac{Q2^2}{Q1^2} & Q2 & (\gamma-1)^{-1} \end{bmatrix} \tag{11}$$

Also the matrix $a = [M]^{-1}[A][M]$ is evaluated as below,

$$a = \begin{bmatrix} u & \rho & 0 \\ 0 & u & \rho^{-1} \\ 0 & \gamma\, p & u \end{bmatrix} \tag{12}$$

The matrix $[a]$ has the following eigen values.

$$\lambda_{1,2,3} = u, u + \sqrt{\frac{\gamma P}{\rho}}, u - \sqrt{\frac{\gamma P}{\rho}}, \tag{13}$$

Where $c = \sqrt{\frac{\gamma P}{\rho}}$ is the wave speed. The eigen-matrix and related theorems are completely derived and proven in Appendix (A).

2.0 Discretization in three dimensions and related benchmark problems

When conservation laws are written in three dimensions, we can follow wxactly similar way we did for the one-dimensional equations to diagonalize the resulting equation. The coefficient matrices and their eigen-values and eigen vectors are fully covered in Refs. [1, 4] and we are not going to discuss them again here. When done, in general form, this method is usually called flux vector splitting in which we have to manage wave amplitude based on the direction of wave propagation. Here we discretize the flux over the edge, and as the simplest approximation, we take the average of flow variables from two nodes forming the edge. This leads to the first-order approximation in space. Two increase the ability of algorithm to handle sharp gradients (such as shocks and irregular geometries) we use

Roe?? averages evaluated at the middle of each edge. Then flux vectors, wave speeds and all transformation and coefficient matrices of conservation laws are evaluated using Roe averages. Integrating the conservation laws over arbitrary control volume V and using the Divergence theorem, simply leads to an update equation for conservative variables at each node versus the distribution of spatial fluxes weighted by the surface area of each element. This equation can be written in the following *semi-discrete* form,

$$\frac{dQ_i}{dt} = R_i\left(t, Q\right) = \frac{1}{V_i}\left(-\sum\nolimits_j \left(F_{ij}.n_{ij}\right) S_{ij} - \sum\nolimits_k \left(F_{ik}.n_{ik}\right) S_{ik}\right) \tag{14}$$

Where $R_i\left(t, Q\right)$ is usually named as residual vector (by steady-state sense of equations) for point i at time t and solution vector Q. The volume V_i is the volume of the i^{th} control volume, j is loop index over the interior, k is the loop index over boundaries, S and F are area and flux vector respectively and n the normal vector for each face all evaluated at previous sections (cell-based grid to node-based grid transformation).

Therefore, if we loop over all edges and evaluate flux vectors using an averaging scheme, then basically we have all requirements to solve Eq. (14) in time. The semi-discrete term $\frac{dQ}{dt}$ changes the numerical representation of PDE equation to ODE formulation thus enables us to apply traditional ODE solver to the problem. It can be simply discretized as $\frac{\Delta Q}{\Delta t}$ resulting a first-order approximation usually known as Euler explicit scheme (Red. [3]). More elaborate discretization may be achieved using classical fourth-order Runge-Kutta scheme which is an explicit multi-stage algorithm fully compatible with the Euler explicit formulation. In fact we repeat the Euler explicit algorithm 3 times between each step and then use a weighted combination of residuals at each point to find the final residual at that point. Mathematical speaking,

$$Qi_{n+1} = Qi_n + \frac{\Delta t}{6}\left(ki_1 + 2ki_2 + 2ki_3 + ki_4\right), \tag{15}$$

Where the weights are obtained using residuals in Eq. (14) as,

$$\begin{aligned}
t_1 &= t_n, Q_1 = Q_n, \\
ki_1 &= R_i\left(t_1, Q_1\right), \\
t_2 &= t_n + \frac{1}{2}\Delta t, Q_2 = Q_n + \frac{1}{2}\Delta t ki_1, \\
ki_2 &= R_i\left(t_2, Q_2\right), \\
t_3 &= t_n + \frac{1}{2}\Delta t, Q_3 = Q_n + \frac{1}{2}\Delta t ki_2, \\
ki_3 &= R_i\left(t_3, Q_3\right), \\
t_4 &= t_n + \Delta t, Q_4 = Q_n + \Delta t ki_3, \\
ki_4 &= R_i\left(t_4, Q_4\right).
\end{aligned} \tag{16}$$

This is proven to be fourth-order accurate. Please note that if we use Euler explicit four times to reach the same computational cost of RK scheme we will get $O\left(1/4\Delta t\right)$ (which is equal to $O(\Delta t)$ for truncation error) and $O(\Delta t)^4$) for Runge-Kutta scheme. Now, assume that we make a refinement in time direction such that $\Delta t_2 = 1/2\Delta t$. In this case which is **frequently encountered in time-accurate solutions** we will get $O\left(\Delta t_2 = 1/2\Delta t\right) = 1/2O\left(\Delta t\right)$ for Euler explicit which is the half of accuracy of original solution. However, for RK we can write,

$$O\left(\Delta t_2\right) = O\left(\left(\frac{1}{2}\Delta t\right)^4\right) = \frac{1}{16}O\left(\Delta t^4\right), \tag{17}$$

Equation (17) clearly demonstrates that **classical Runge-Kutta scheme is sixteen times more accurate than Euler explicit when time step is divided at the middle.** This generic conclusion proves that RK method has great advantages when **time-accurate** solution of conservation laws is sought. In the next section we proceed to solve sample benchmark problems.

2.1 A Cube Meshed with Tet and Hex elements

To validate the algorithm specially the ghost-edge based **far field** boundary conditions, we proceed to solve a very basic and descriptive problem which is best suited for debugging purposes. In this problem, we consider a cube which is meshed with tetrahederal and hexahederal elements. We consider a far field flow of Mach number 0.5 (subsonic) coming toward the cube. As we know from the theory, there shouldn't be any change in the initial value of the field *if all cube sides are set to be far field.* Therefore if the solution is not converging or we get non-physical solution, this is the best place to go over the code and find the errors that might be hidden in the code. We performed numerical simulation for flow of different directions $u = 0.5$, $v = 0.5$ and $w = 0.5$ and a combination of these for a fair interval of CFL numbers $CFL = 0.5$, $CFL = 0.75$ and $CFL = 1.0$ for both Hex and Tet element grids. The results are brought in fig.(**??**). As shown for hex mesh, the residuals is usually a constant line since this is a perfect grid in respect of symmetrical issues and therefore the cell-based to node-based transformation works perfectly (which sum of the area vectors exactly equal to zero). Therefore nothing remains as the grid transformation residuals to be shown-up in the solution residuals. However, for the Tet grid we observe that there are oscillations in the computed residuals which are always bounded even for CFL =1.0. The interesting point is that if the flow direction is not perpendicular to the hex mesh, oscillations in the residuals can be observed.

2.2 Supersonic Ramp

This problem is solved on the unstructured grid shown in Figs.(8). Supersonic flow of $M_\infty = 2.$ and $\rho_\infty = 1.$ (in non-dimensional form of equations) is applied for free stream condition. The free stream cross components of velocity vector, i.e. v and w are assumed to be zero. Free stream boundary conditions are applied to the inlet face and outlet face while

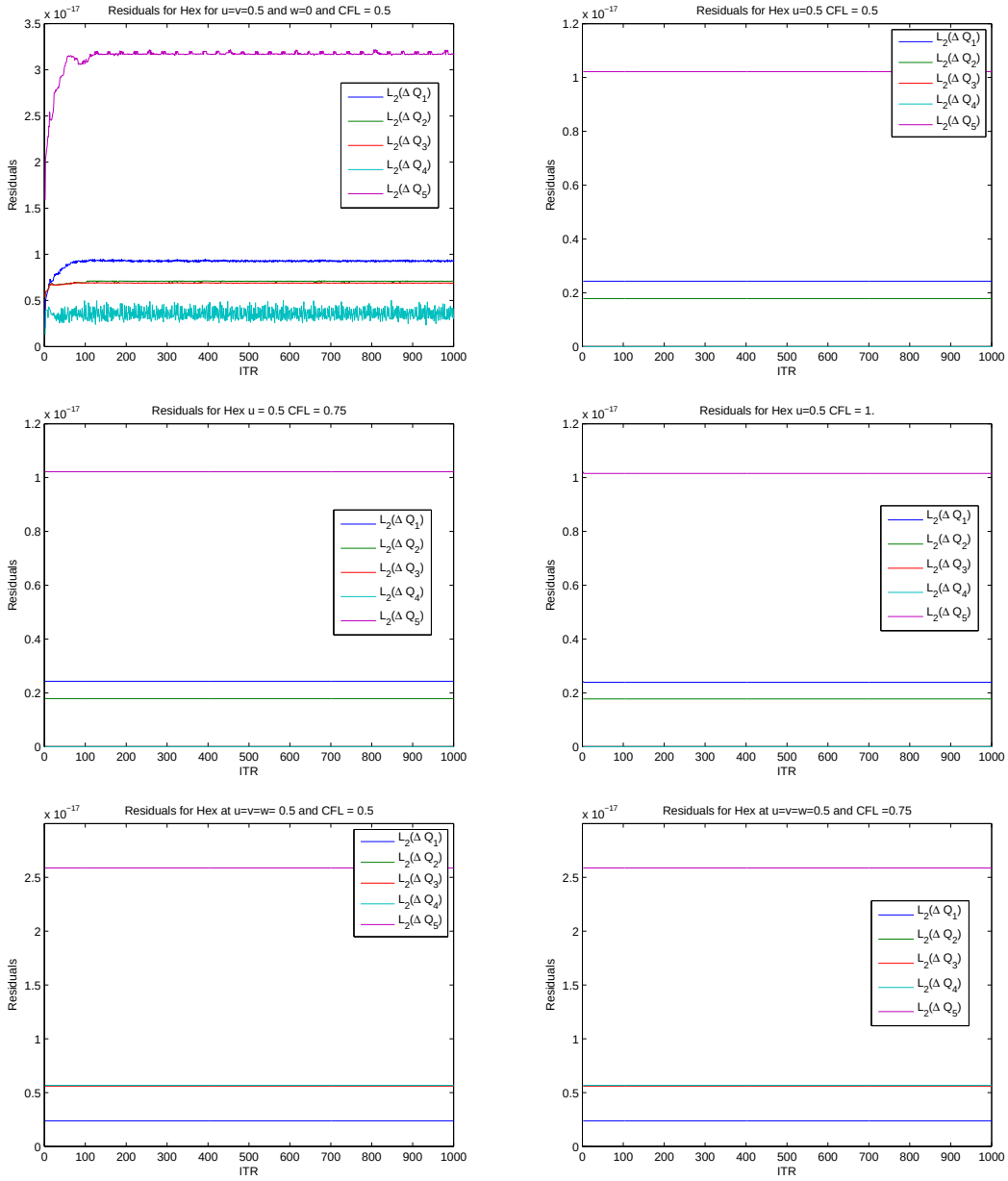


Figure 4: Residuals for cube meshed with tet and hex for variation combinations and CFL numbers and combinations thereof.

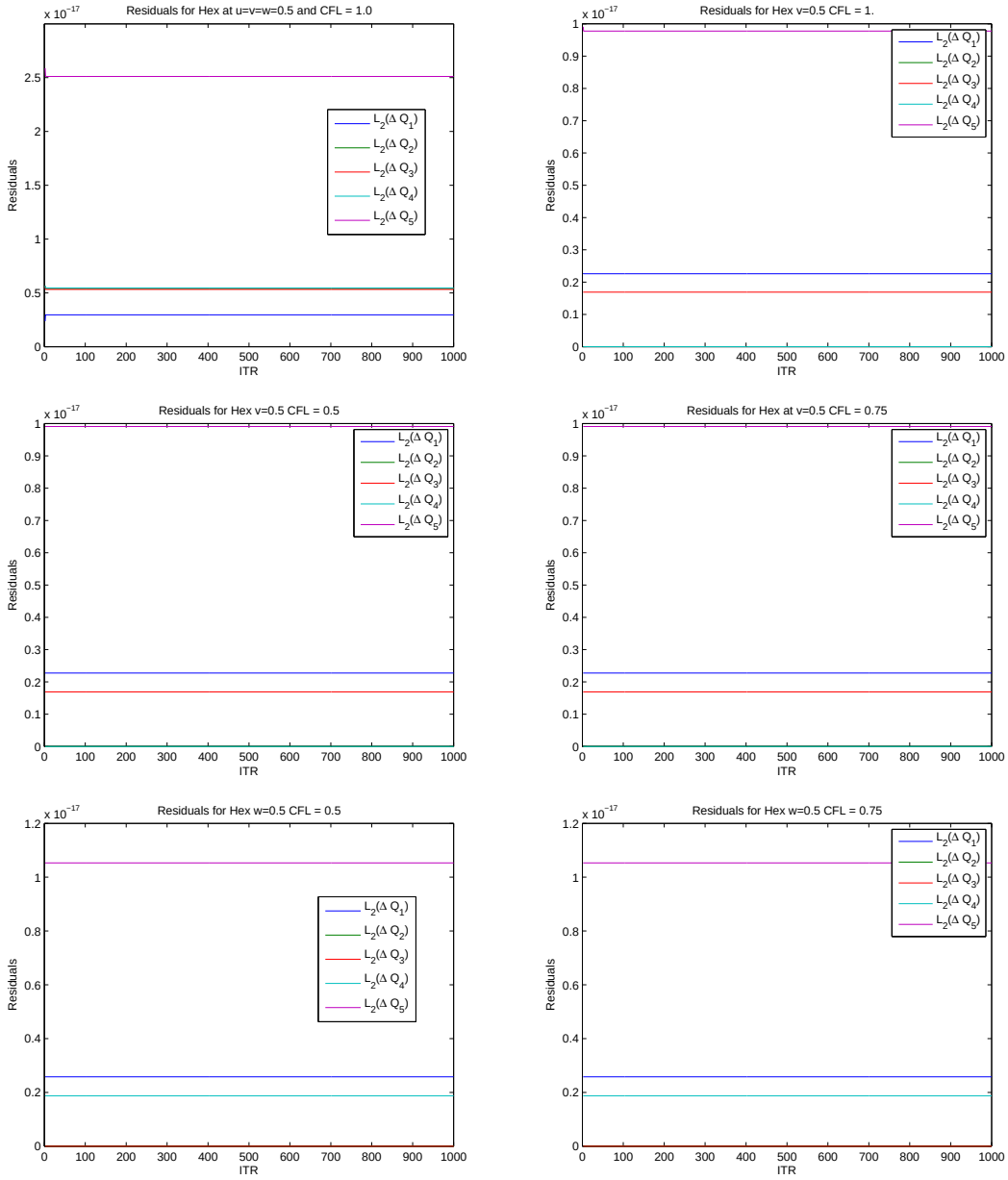


Figure 5: Residuals for cube meshed with tet and hex for variation combinations and CFL numbers and combinations thereof. (continued)

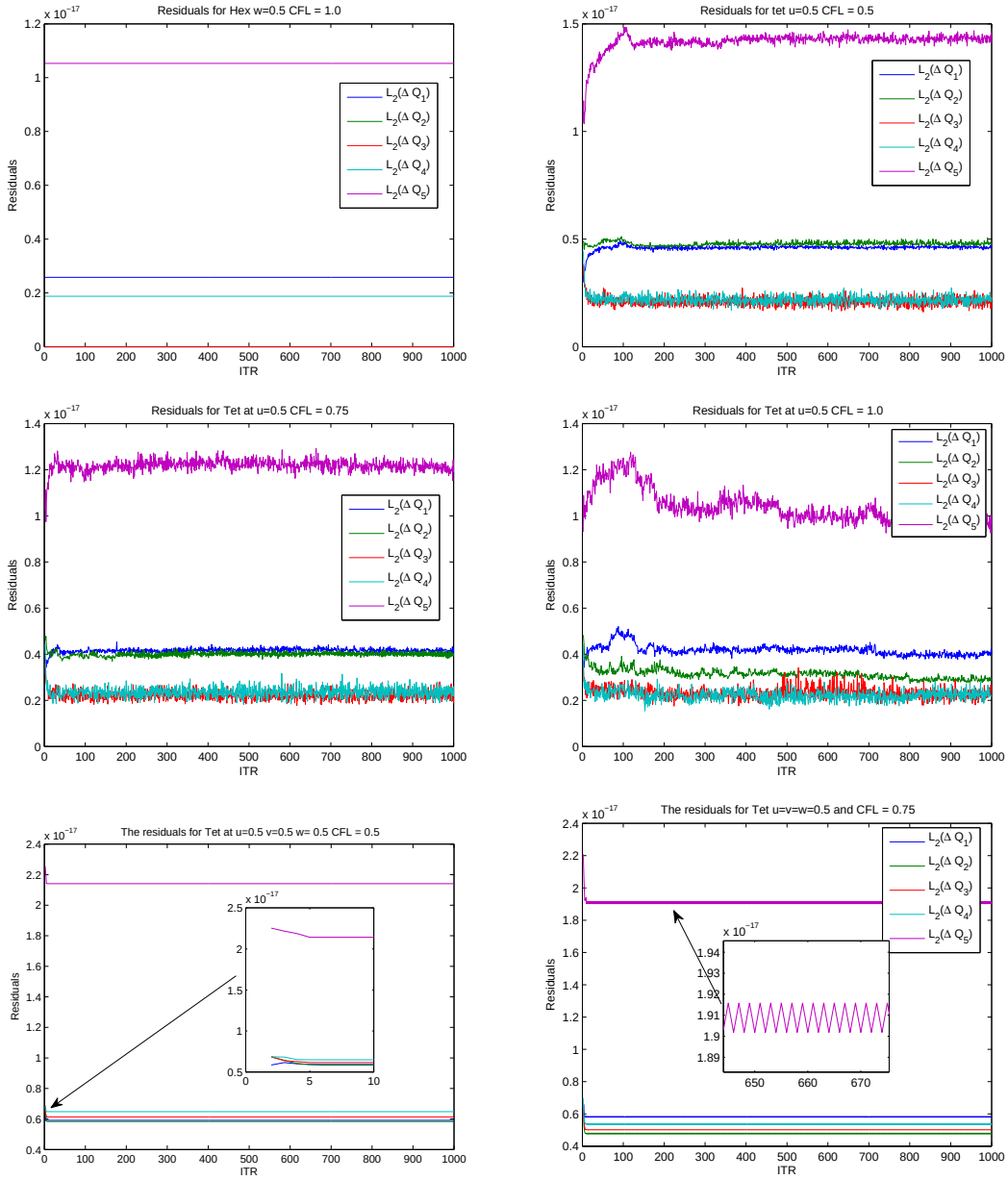


Figure 6: Residuals for cube meshed with tet and hex for variation combinations and CFL numbers and combinations thereof. (continued)

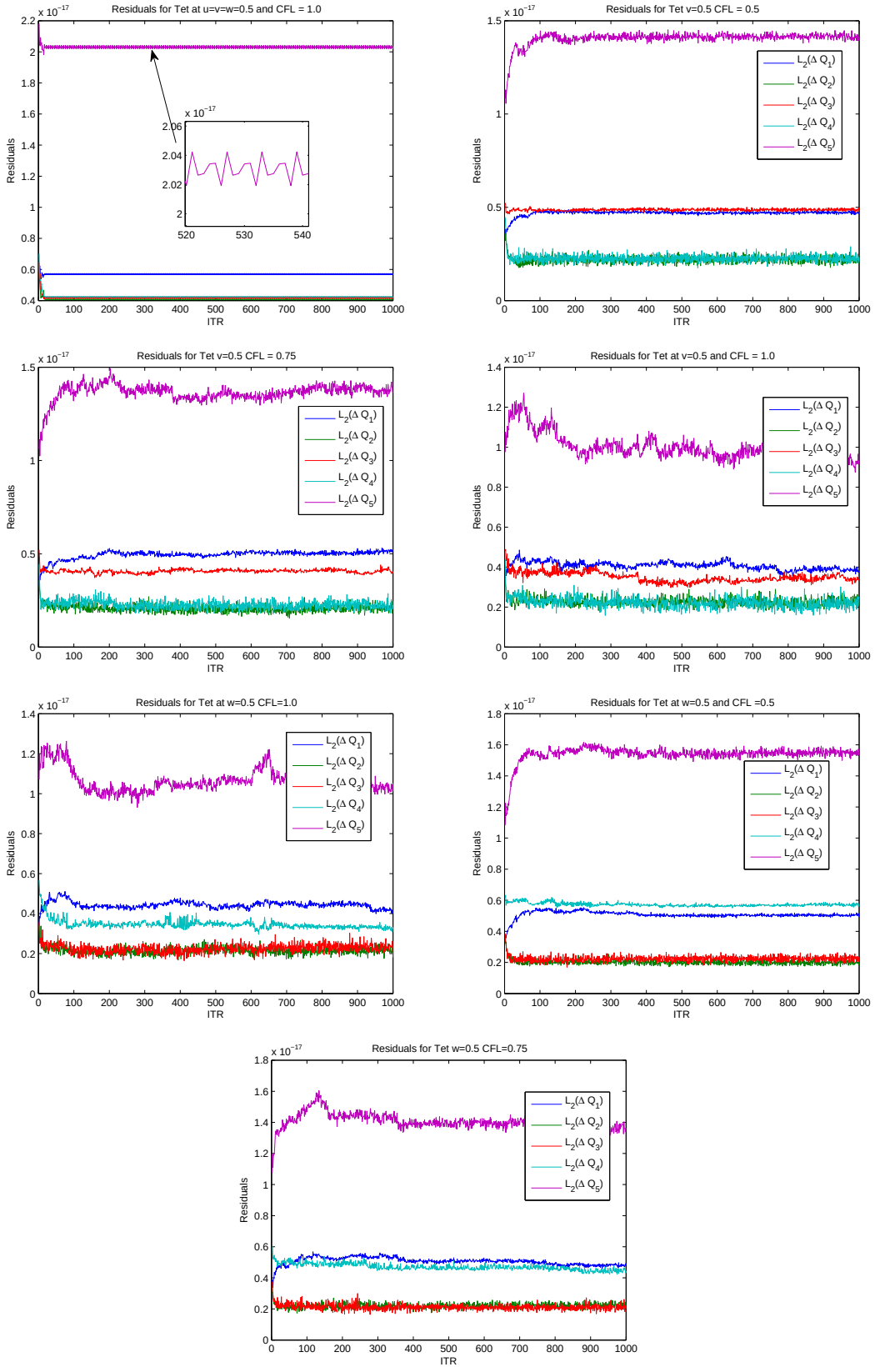


Figure 7: Residuals for cube meshed with tet and hex for variation combinations and CFL numbers and combinations thereof. (continued)

the rest of boundary is assumed to be inviscid walls. Different implementations of boundary conditions are done. For example, we assumed that the side faces are free-stream and solid wall separately. Solving the equation leads to the same steady-state solution for both cases. We used a couple of different initial conditions and all lead to the same steady state solution. The initial we used for convergence curves shown in (10) are given in Ref.([1]) and we don't mention it for brevity. Using the coarse grid, the equations are solved with $CFL = 0.7$. The final converged solution is presented in fig. (8). It is evident that the overall structure of solution is correct because a shocked is resolved over the ramp and it passes the outlet without any reflection. Also no sawtooth wave appears at the inlet or side faces.

However the shock is not as thin as what we expect from the oblique shock theory in Aerodynamics. It seems that the shock is distributed over multi cells so that each cell has a contribution to the shock region. This gives us the idea of concentrating mesh near the shock.

The convergence study of coarse grid ramp problem is presented in fig.(10) and fig.(11). The residuals presented in this report are based As shown, the solution is converged after 10500 iterations. To make sure that the discretization is fully consistent, we continued running the program after convergence and it can be seen that the residuals remain near machine truncation error for a long time after solution converged. So we are sure that the entire discretization (interior plus boundaries) is consistent because otherwise an inconsistent converged solution may make chaotic instabilities due to positive eigen-values arising from inconsistency and in that case the residuals may start to increase and go to oscillatory behavior. To make sure that all solution components are correct, wrote all five residuals to hard disk and the results of post processing are shown in fig.(11) and in fig.(12) for CFL=0.8. As shown all residuals are near machine zero after 12000 iteration.

The next thing that we suspect regarding the validity of solution is the shock angle. This is very important because in the eigen-valued based algorithms if the wave speed calculation

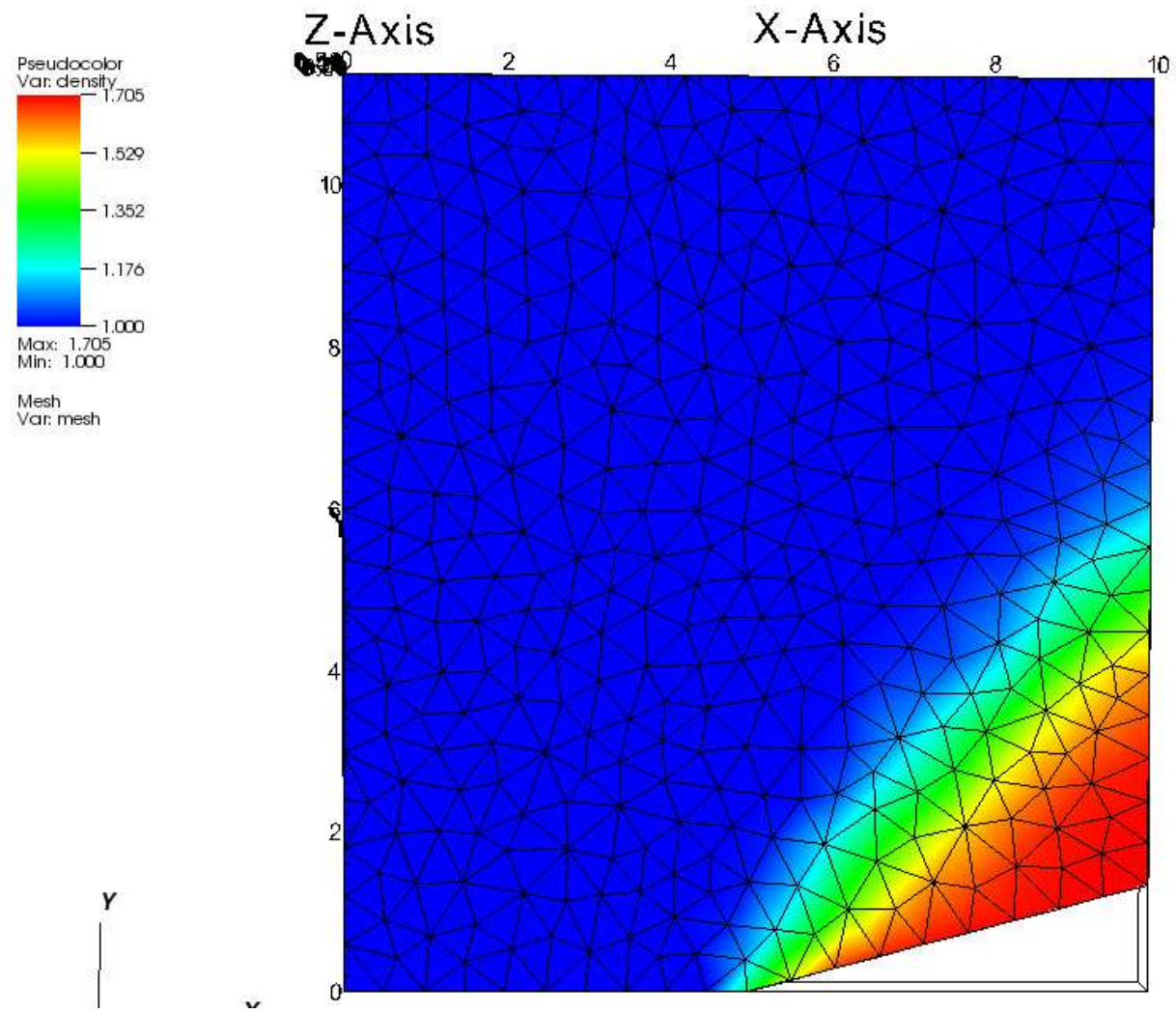


Figure 8: The density contours of converged solution for coarse mesh. The shocked is resolved over multipile cells.

is not correct then the algorithm may converge to a non-physical solution and hence as a result the angle of shock may be different from theoretical and experimental values. Here we have two options to calculate shock angle. The first option is to find the average of all cells in the coarse grid contours of *pressure* or *density* where the shock is resolved. Connecting the averaged coordinates, we can easily *estimate* the shock surface in three dimensions.

The second option that we have is to use a refined grid in which near shock region is highly refined. In fact we used the former approach to find the shock angle and properties in this work. The refined mesh is shown in fig.(14). The free stream and boundary conditions are

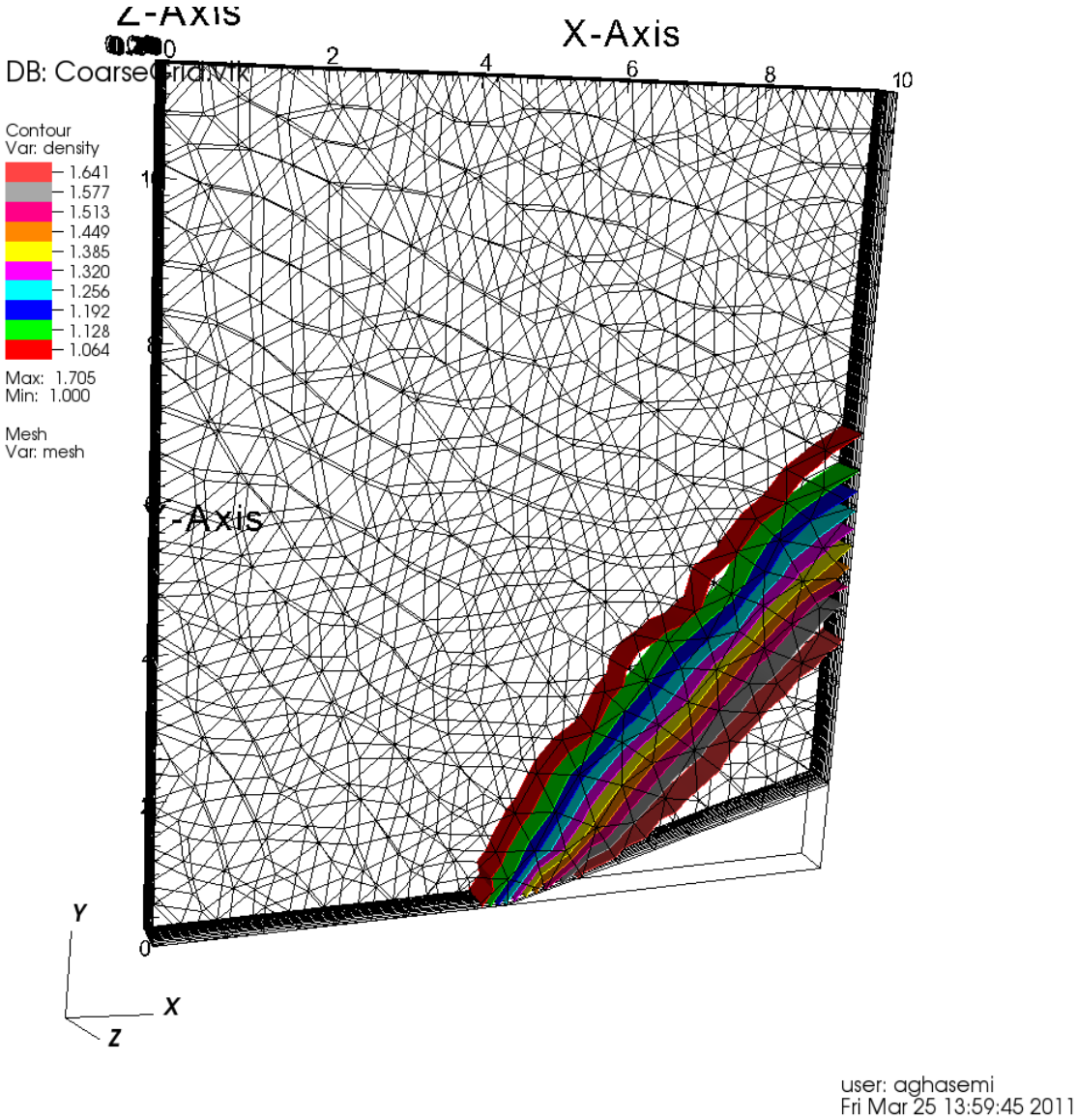


Figure 9: The constant density surfaces of converged solution for coarse mesh. This clearly visualizes that the shock is not resolved physically.

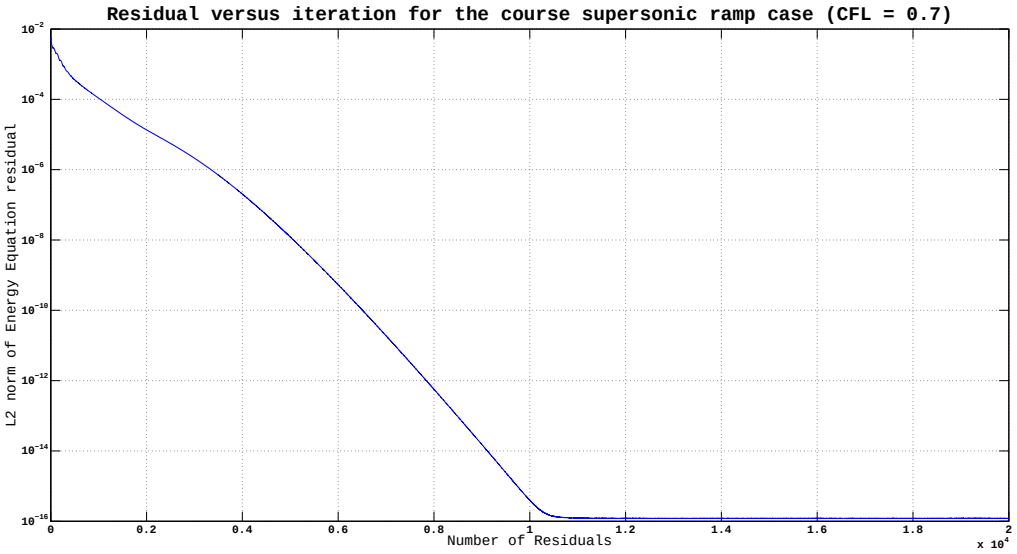


Figure 10: The energy equation residuals versus iteration. We separated energy equation convergence because it usually converges slower than other equations.

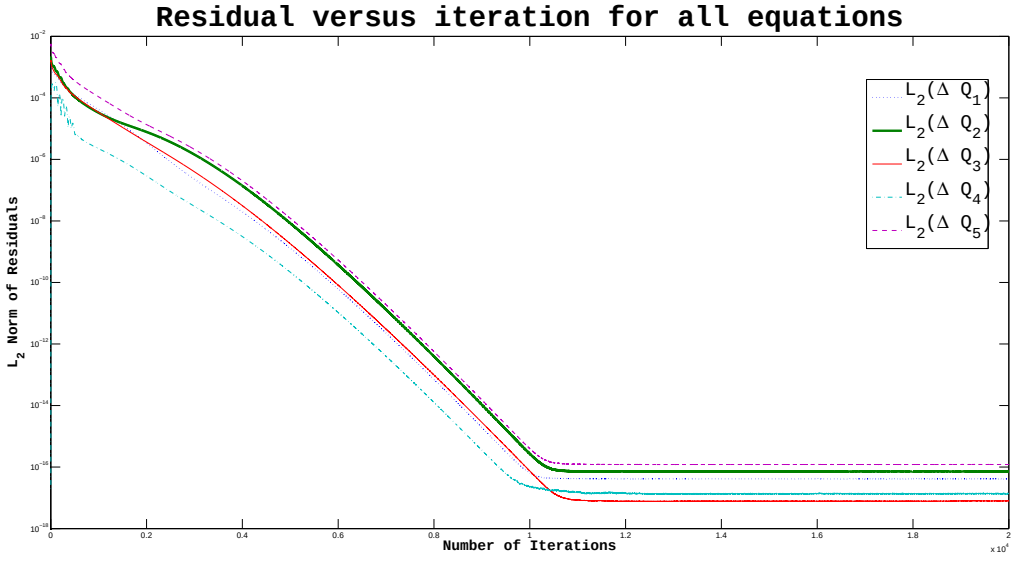


Figure 11: Residuals for all equations, continuity, momentums (x,y,z) and the energy equation. CFL = 0.7

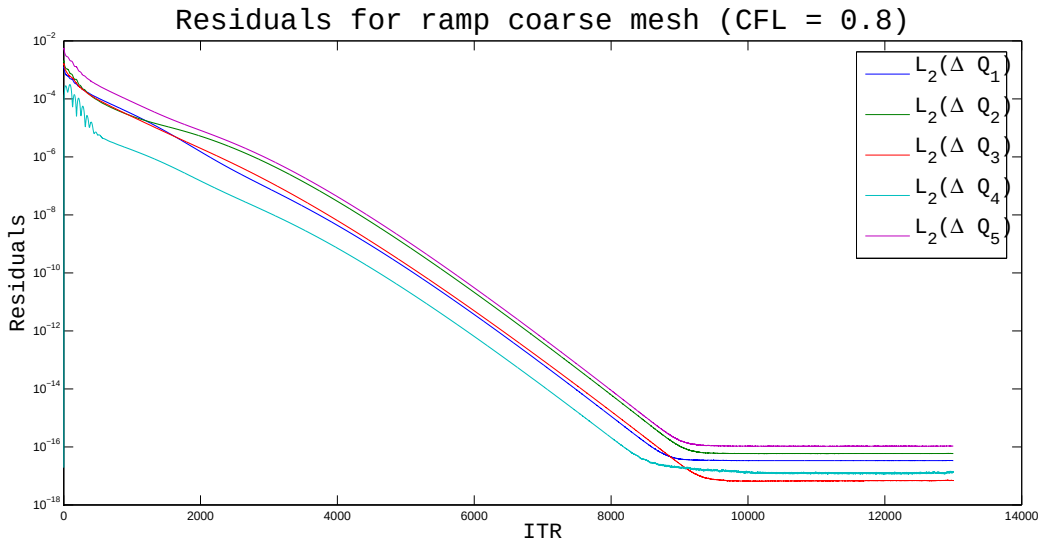


Figure 12: Residuals for all equations, continuity, momentums (x,y,z) and the energy equation. CFL = 0.8

applied the same in this problem. As shown in (14) the shock is perfectly captured and the computational domain is divided into two distinct parts. The one before the shock and the one after the shock. This is exactly what we expect from the quasi-one dimensional theory.

The contours of total energy per unit volume are plotted in fig.(15). The pattern follows the density behavior as we expected.

The convergence study is shown in fig.(16) and (17) for CFL = 0.8. It is concluded that the solution converges slower than coarse grid solution. This is mainly because smaller grid size h appears in the time step calculation subroutine. Once again we run the program after the converges and no nonlinear instabilities observed for refined mesh.

2.2.1 Alternative time-marching scheme

The results presented in the previous section were obtained using explicit Euler stepping. However, we are interested to apply high-order time discretization schemes because as mentioned before, they play essential role in time-accurate simulation. In this case, Runge-kutta

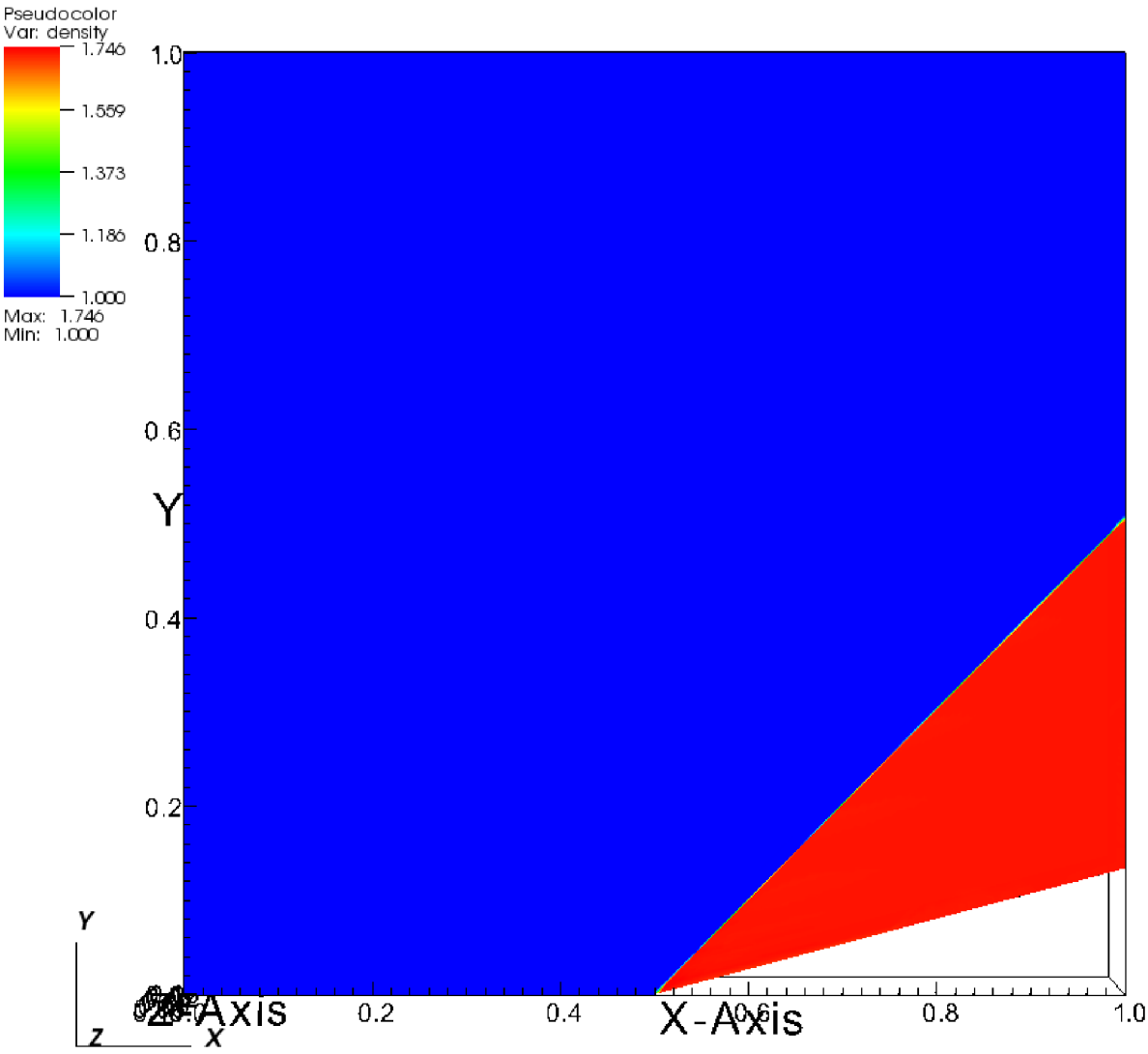
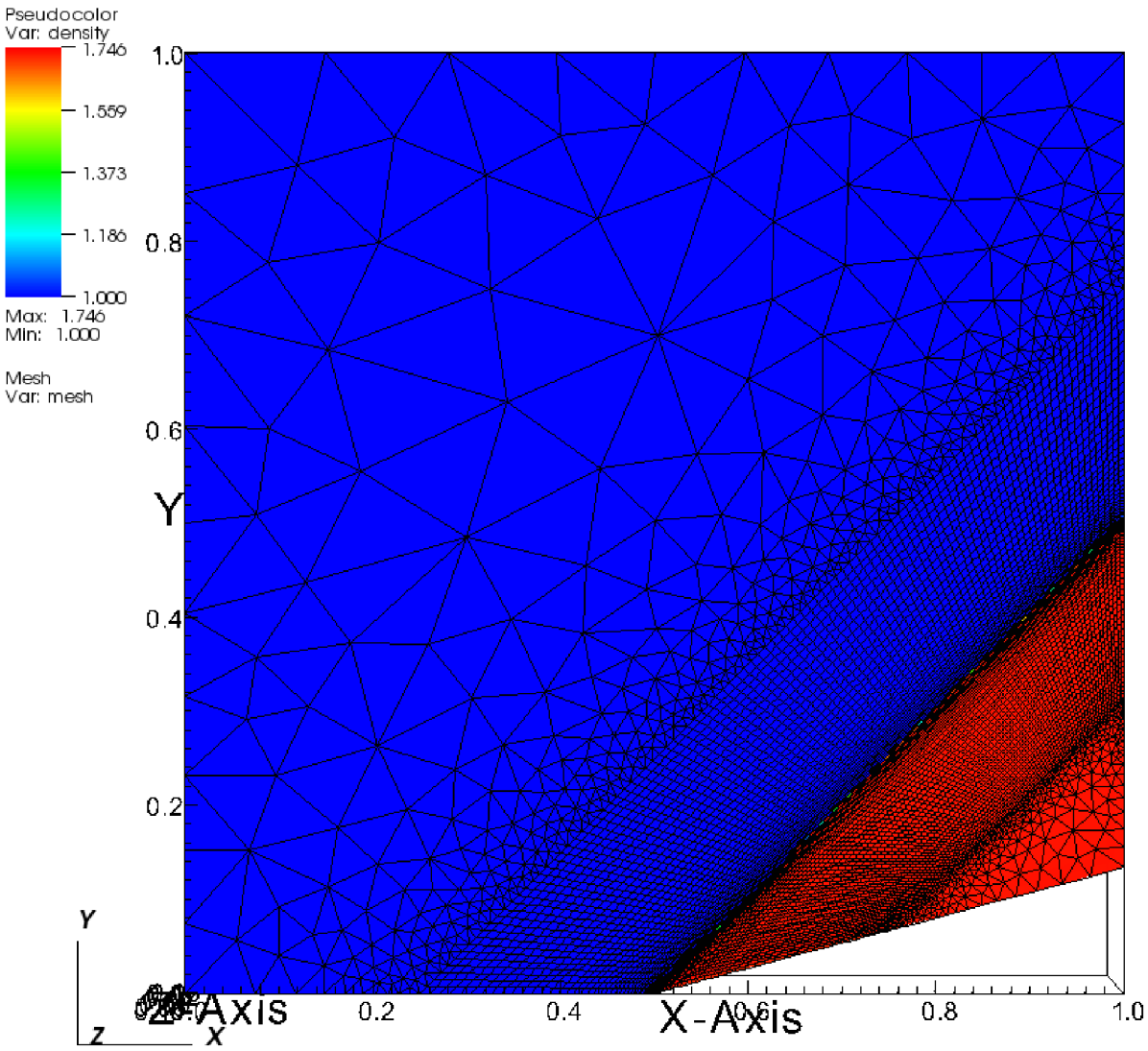
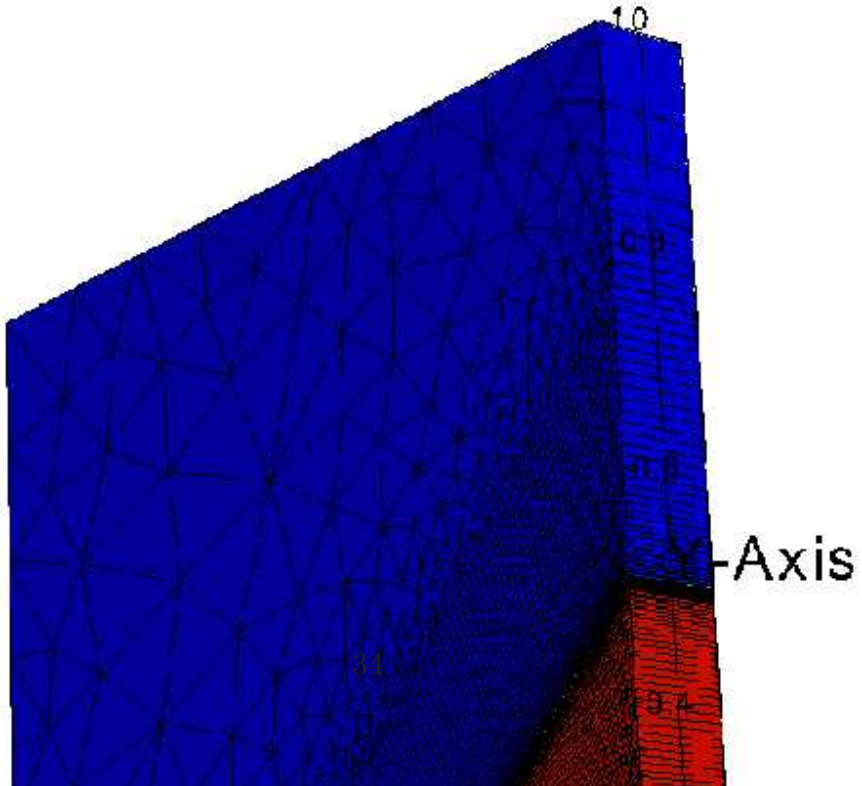


Figure 13: Density contours for refined grid. The inlet mach number is $M_i = 2.0$.


Pseudocolor
Var: density
1.746
1.559
1.373
1.186
1.000
Max: 1.746
Min: 1.000
Mesh
Var: mesh
1.0
0.8
0.6
0.4
0.2
Y
Y
Z
X
Axis
0.2
0.4
X-Axis
0.8
1.0


1.0
-Axis

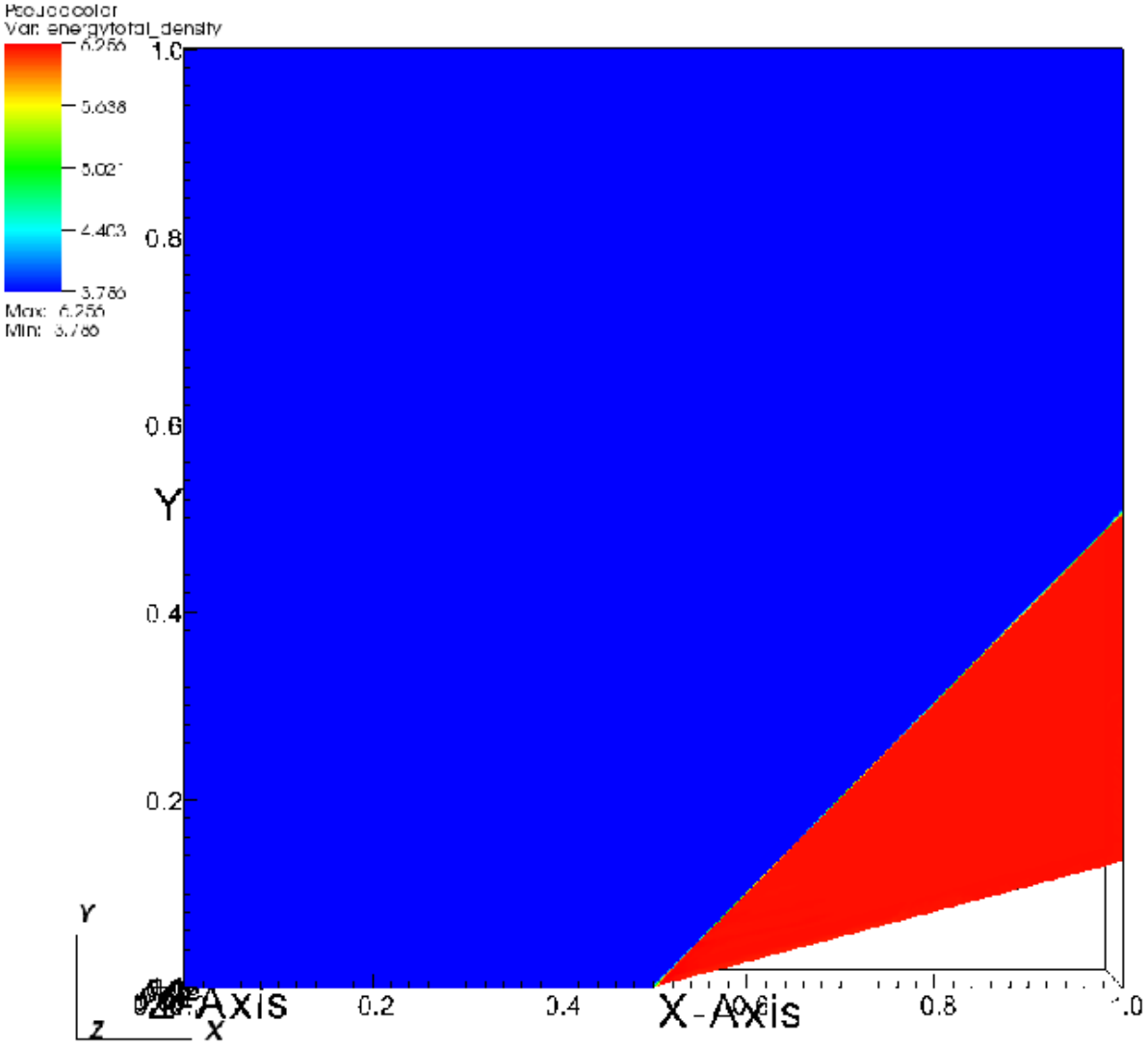


Figure 15: The contours of fifth conservative variable (ρe_t).

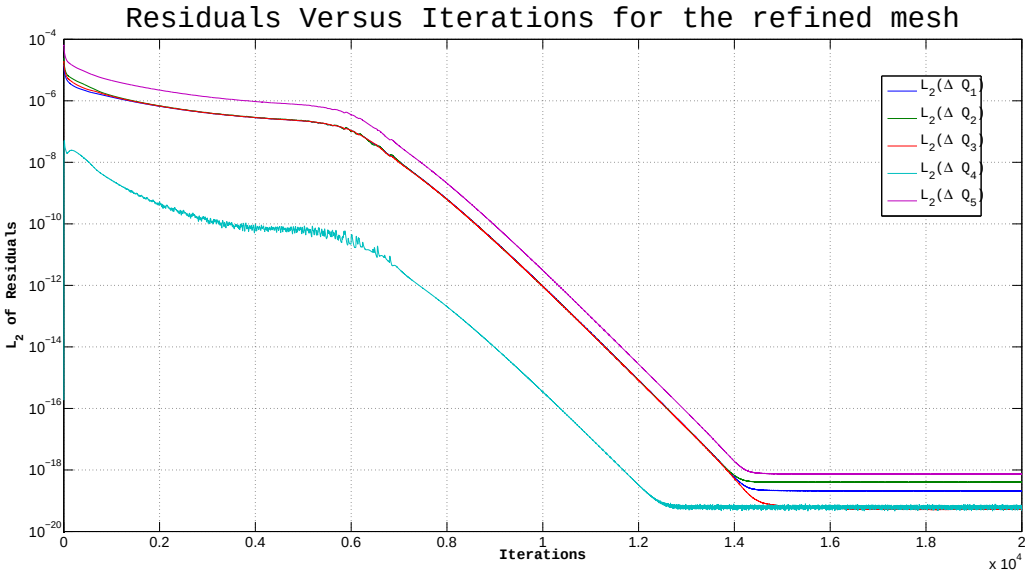


Figure 16: All residuals for the case of refined mesh. CFL = 0.7.

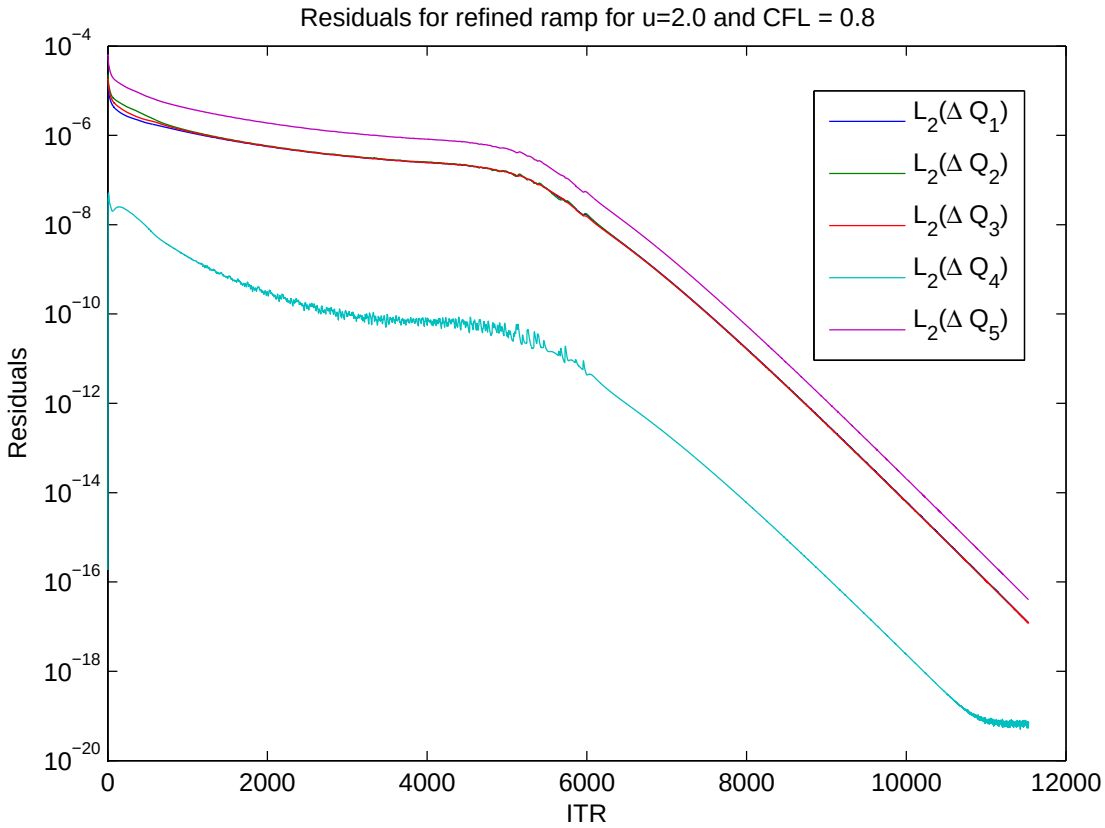


Figure 17: All residuals for the case of refined mesh. CFL = 0.8. We zoomed the residuals to show that ITR = 12500 for CFL=0.7 (fig.(16) reduces to ITR= 11000 for CFL = 0.8 for the w momentum equation to be converged.

family of high-order time marching scheme provide arbitrary accuracy (depending on the number of stages and CFL number) in the time direction while it is extremely easy to implement. However we should always bear in mind that increasing the order of accuracy of temporal scheme will not necessarily increase the **overall** order of accuracy of discretization, since the latter depends on the order of accuracy of *temporal and spatial* discretization. In the current algorithm, the spatial terms are evaluated using linear averaging (interpolation) which remains a first order truncation term $O(\Delta x)$ in the *fully discretized* form of scheme. This term is always dominant for **coarse** grids compared to high-order terms $O(\Delta t^n)$ arising from time-discretization, therefore the **overall** order of the scheme is one. Here to investigate the validity of the algorithm, we implement *both classical RK4 scheme (eq. (15)) and five stage low-storage scheme of Jameson and Mavriplis Ref.([].* The latter is particularly suited for implementing multi-grid remedies for accelerating convergence which might be the extension to the current work. The fifth-stage RK scheme is simply implemented as follows. After initialization of the field, we compute the initial residuals which are used to calculated required time-step (based on the eigen-value computation done in Residual calculation module). Then we go through five-stage RK loop and update the conservative variable according the following equations.

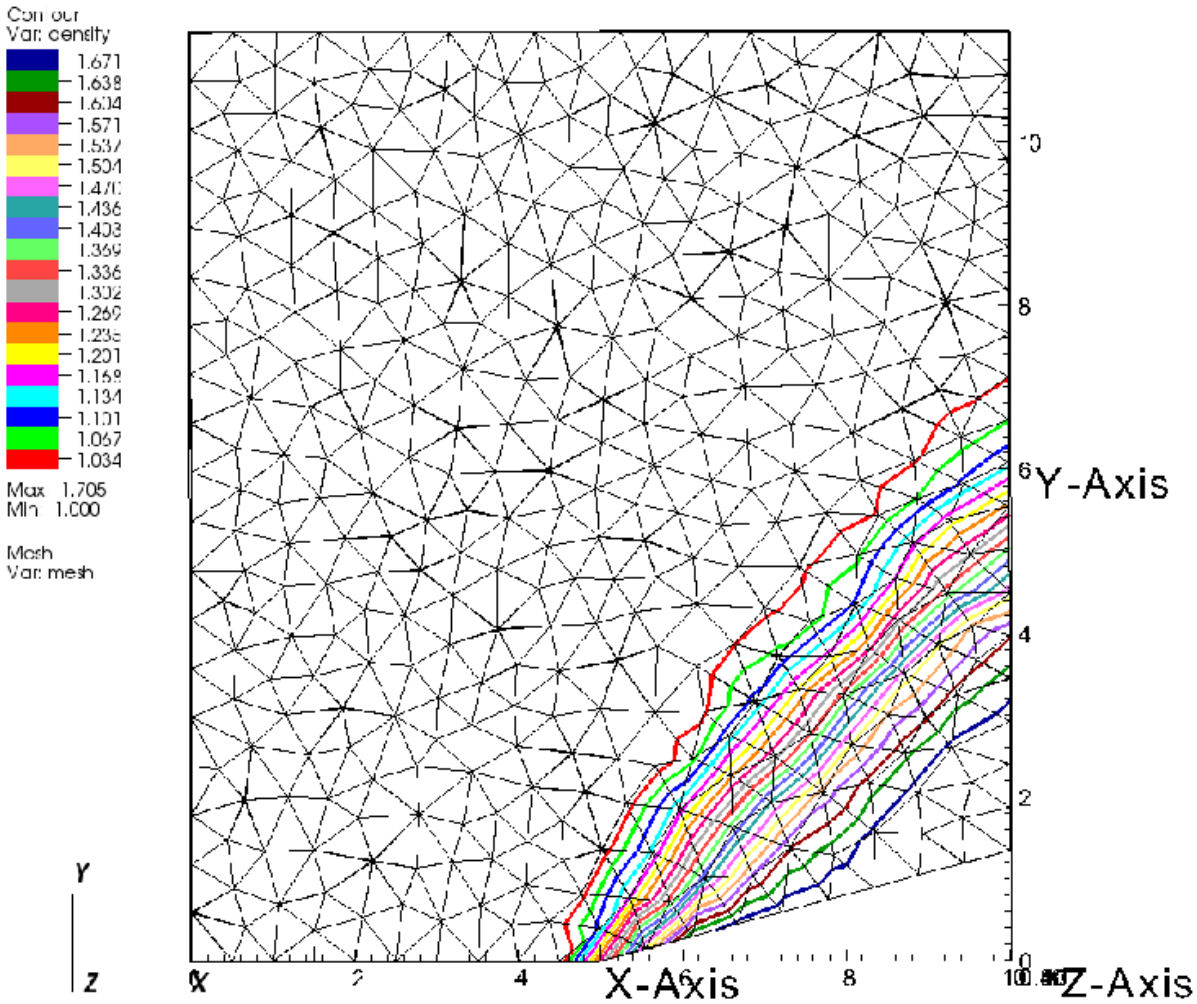
$$\begin{aligned} w^{(0)} &= Q_n, \\ w^{(1)} &= w^{(0)} - \alpha_1 \Delta t / VR\left(w^{(0)}\right), \\ w^{(2)} &= w^{(0)} - \alpha_2 \Delta t / VR\left(w^{(1)}\right), \\ &\vdots \\ w^{(5)} &= w^{(0)} - \alpha_5 \Delta t / VR\left(w^{(4)}\right), \\ Q_{n+1} &= w^{(5)}. \end{aligned} \tag{18}$$

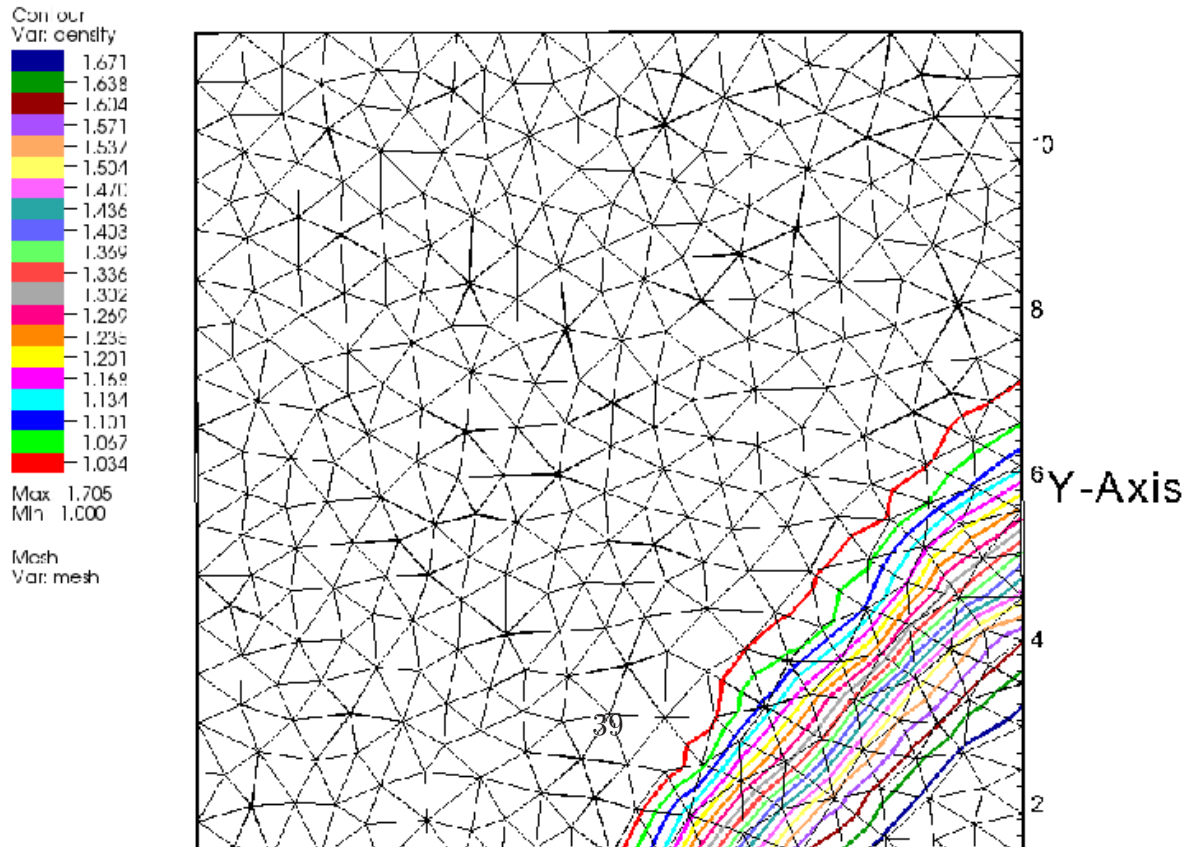
boundary conditions are applied after the last stage of RK5 sub iteration. The the process is repeated until a predefined number of iterations. We could implement a condition for convergence, nut we intentionally didn't do that because we wanted to investigate the stability behavior of the scheme after convergence. The result of steady state solution are presented in fig.(18). It is apparent that the spatial distribution of contours are the same because we used first-order approximation for fluxes in all cases. So after vanishing the temporal term, we expect to achieve to the same result.

The convergence study is performed in fig.(19). For classical Runge kutta we increased the CFL condition so that the best convergence achieved. As shown the convergence is accelerated in both RK schemes compared to explicit Euler stepping (although all lead to the same converged solution as discussed before). Summarizing this section we conclude that Runge- Kutta schemes provide accurate result while they are able to improve convergence since we have wider space for CFL choice.

2.3 Comparision of ratios with the one-dimensional oblique shock theory

There is a well-know nonlinear one-dimensional oblique shock theory which is obtained by taking a control volume along the shock so that the velocity before and after the shock are projected in the normal directions. If we write one-dimensional continuity, inviscid momentum and energy equations for the control volume, and substitute the equations into each other and do some simplifications and rearrangement, we reach to the *one-dimensional oblique shock equations* which express the shock angle, pressure ratio, density ration, temperature ratios in the terms of wedge angle and free stream mack number. These relations are in analytical forms however to simplify calculation which can be done by a very crude calculator, some resources have developed tables and chart which express the same relations.


Con our
Var: density
1.671
1.638
1.604
1.571
1.537
1.504
1.470
1.436
1.403
1.369
1.336
1.302
1.269
1.235
1.201
1.168
1.134
1.101
1.067
1.034
Max 1.705
Min 1.000
Mesh
Var: mesh
Y
Z
X
2
4
X-Axis
8
10
0
2
4
6
8
Y-Axis
Z-Axis


Con our
Var: density
1.671
1.638
1.604
1.571
1.537
1.504
1.470
1.436
1.403
1.369
1.336
1.302
1.269
1.235
1.201
1.168
1.134
1.101
1.067
1.034
Max 1.705
Min 1.000
Mesh
Var: mesh
30
0
2
4
6
8
Y-Axis

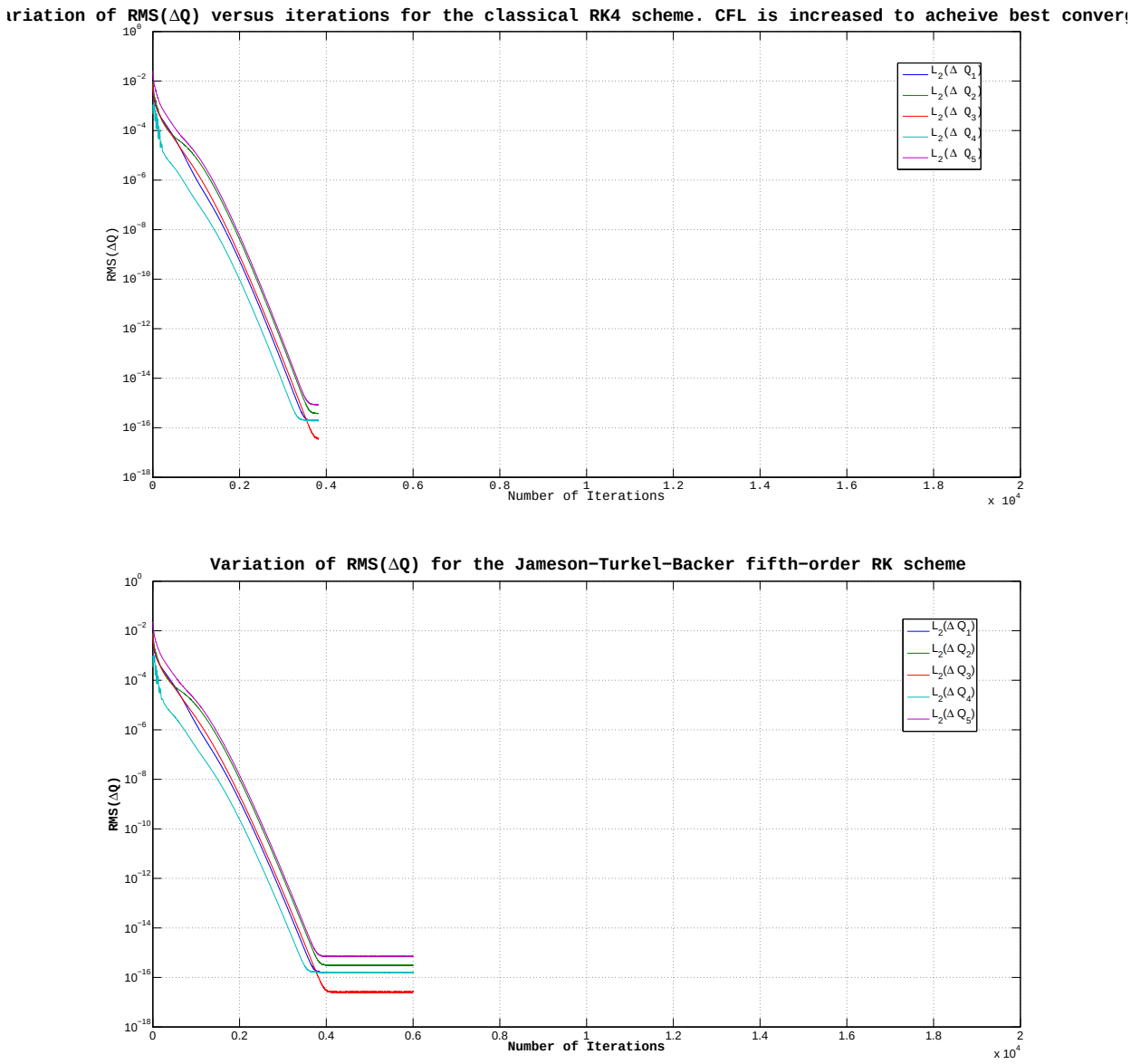


Figure 19: The converged solution obtained using different Runge-Kutta type scheme. (Top) classical RK4. (Bottom) The Jameson-Turkel-Backer fifth order scheme.

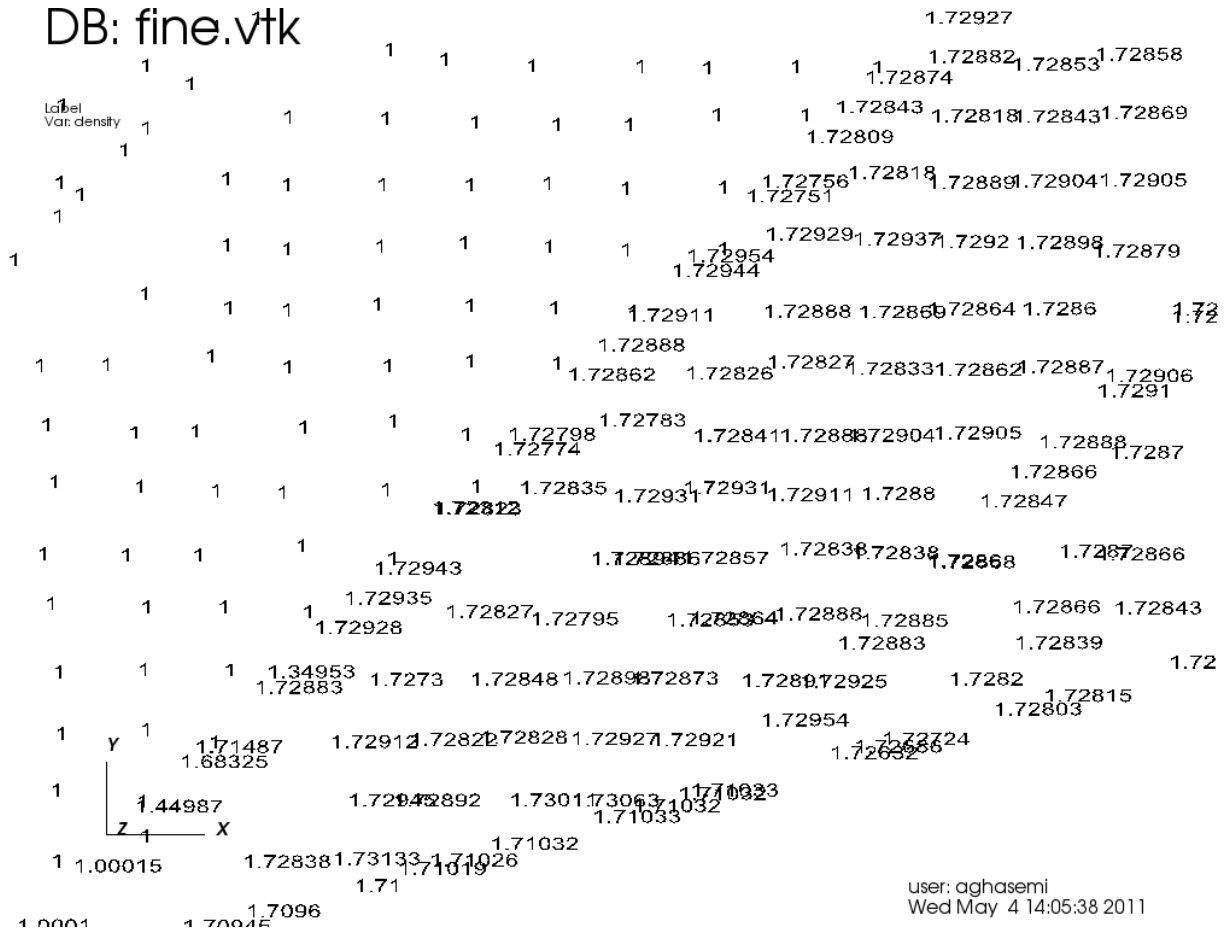


Figure 20: The dumped solution near the oblique shock. There is a negligible difference in the computed solution. We just get the average value.

Here we use the equations in the John Anderson's Aerodynamics book (Ref(**??**)). Doing so for the wedge angle $\theta = 15^o$ and free stream Mach number $M_\infty = 2.0$ we get $\rho_2/\rho_1 = 1.732$ and $p_2/p_1 = 2.183$. Dumping the solution near the shock we find that there is a little difference between solution after the shock. This is clearly shown in fig.(20). We took some values and the average is $\rho_2/\rho_1 = 1.729$.

Also for pressure we get the average of $p_2/p_1 = 2.190$. These are compared in table (6) below.

The reason that the refined mesh gives a solution very close to the oblique shock theory is that **since we refined mesh near the shock, the assumption of a one-dimensional**

Parameter	Theory	Computed	Percent of Relative Error
ρ_2/ρ_1	1.732	1.729	0.17/100
p_2/p_1	2.183	2.190	0.032/100

Table 6: The comparison between theory and numerical solution.

control volume normal to the shock becomes numerically implemented. Therefore we expect that the solution obtained by mesh refinement must also be the same with the control volume based theory. However as described in more details before, the coarse mesh cause the shock to be resolved over multiple cells. Therefore some cells (or numerical dissipation error) might act as an **strong expansion wave region** which reduce the total strength of the shock.

2.4 Three-dimensional nonlinear acoustical resonator

The formation of nonlinear standing waves in a gas-filled closed cavity has been the focus of many theoretical and practical investigations. Consider an axisymmetric tube of length L and radius R(x) as the target resonator. Acoustic source for generating oscillating flow is provided by the vibration of entire resonator or a flat piston with a periodic acceleration. The tube is closed and contains an ideal gas initially at rest (See Fig.(1)). Depending on the frequency and amplitude of oscillations, geometry of resonator and properties of the gas, this problem encompasses a variety of physical aspects including traveling shocks, turbulence and heat transfer. For the case of a long narrow tube oscillating outside of the shock region, Merkli and Thomann derived a first-order analytical (perturbed) solution for laminar fields and a second-order (time-averaged) solution for heat transfer. They found that for small values of Prandtl number, thermoacoustic effects become dominant. In a pioneering experiment, Merkli and Thomann observed transition in Stokes layer at critical Reynolds $Re_\delta = 400$ (based on the thickness of Stokes layer). They found that complex turbulent bursts decay by a kind of relaminarization process in the same period of oscillation.

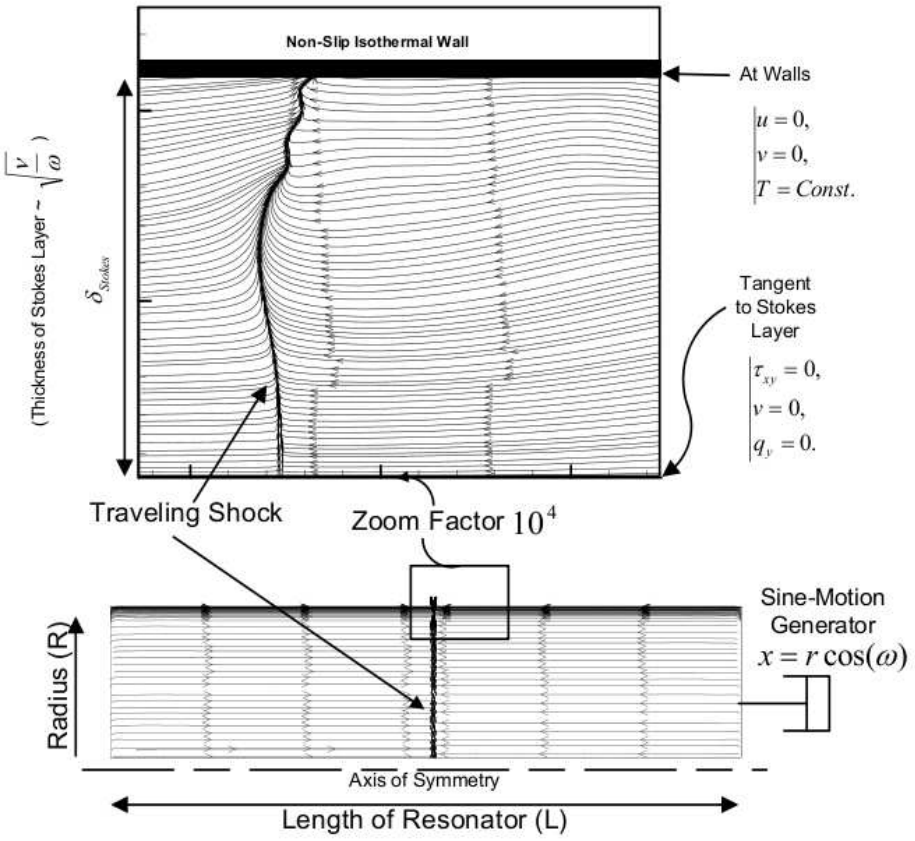


Figure 21: A typical shock/stokes-layer interaction (bottom). Near-wall region including the traveling shock is 10^4 times magnified (top). The entire closed resonator is oscillated by cylinder-piston mechanism shown in the figure.

The numerical and analytical solutions used so far are derived using simplified forms of Navier-Stokes equations and are suitable for particular purposes. Theses methods are strongly dependent to the simplifying assumptions that restrict the range of applicability of numerical algorithms for real practical problems of nonlinear acoustics and thermoacoustics in closed cavities including large-amplitude oscillations, heat transfer and turbulence where the assumptions of isentropic compression and negligible boundary-layer are evidently invalid.

To remove such limitations, full compressible two-dimensional Navier-Stokes equations are employed to directly investigate the nonlinear standing waves inside closed cavities. However, there are many limitation in the finite-difference scheme when applied to three-dimensions. These include lack of 3D filters, singularity in corner points and limitation regarding complex three-dimensional geometries. therefore in this work we apply the current first- order finite volume scheme to only validate formation of shock. In the case of agreement, this work will be continued until gih-order accuracy in space and time are acheived. The final solver will enable us to perform three-dimensional direct numerical simulation (DNS) of Arbitrary-Reynolds nonlinear acoustic field in a shaped cavity which leads to the understanding of fine scale turbulent Stokes layer and its complex interaction with traveling shocks (See Fig.[1]) and probably provides a theoretical explanation for observations of Merkli and Thomann.

In the current simulation we assume that a piston is inserted at the end of cylinder. The test experimental setup and results are shown in fig.(23) and fig.(24) respectively. Here we used zero initial conditions for velocity vector and initial pressure P_0 for the fluid inside the cavity. A very coarse three-dimensional grid is generated for the cylinder geometry which is shown in fig. (22).

The inlet condition is imposed as a perodic piston (wall) with velocity $u = A\omega \cos(\omega t)$.

The solution does not have a converging residual behavior. Instead residuals go to periodic oscillations. The mean value of pressure oscillations increase from P_0 and goes to a stady

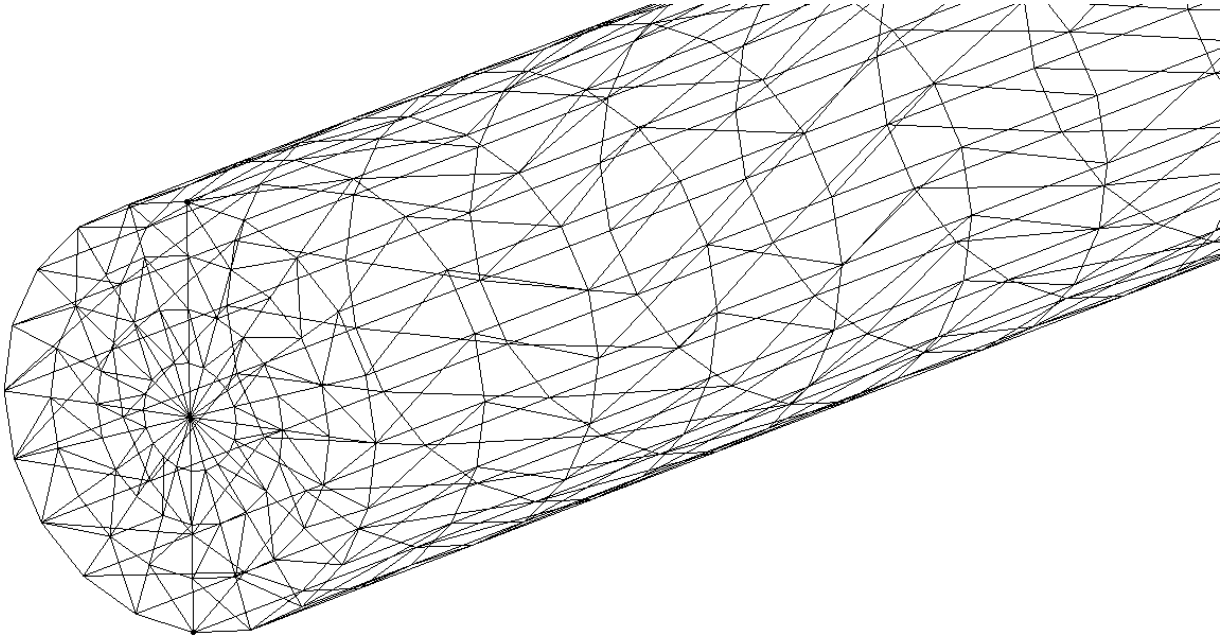


Figure 22: Coarse mesh used for cylinder simulation.

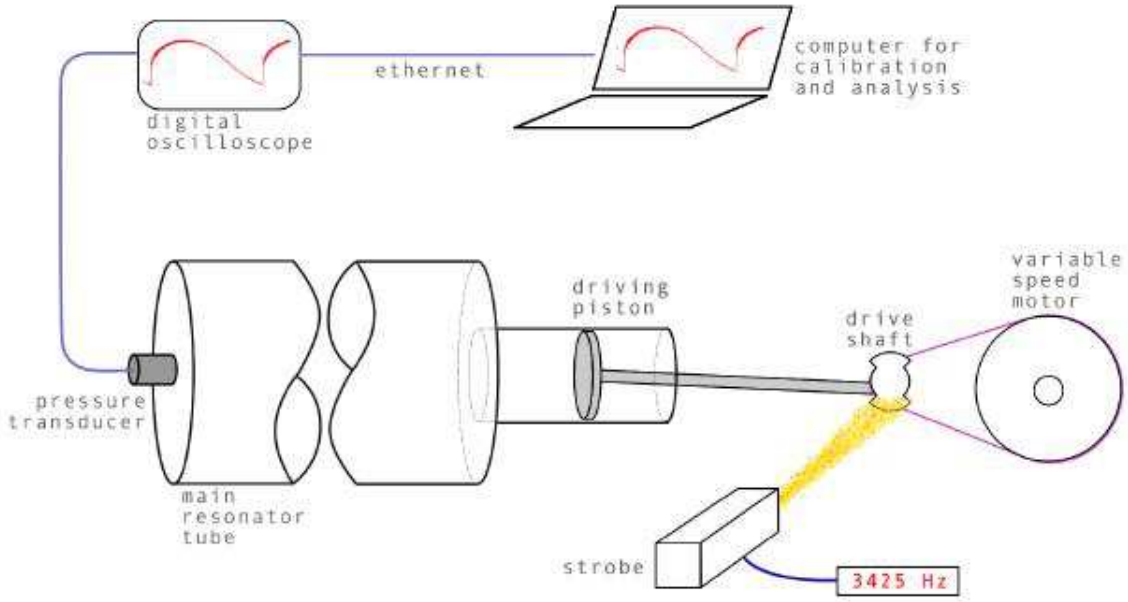


Figure 3.1: Schematic of the experimental apparatus.

Figure 23: Experimental setup for nonlinear tube acoustic problem. A cylinder is inserted at the opening of a closed rigid tube.

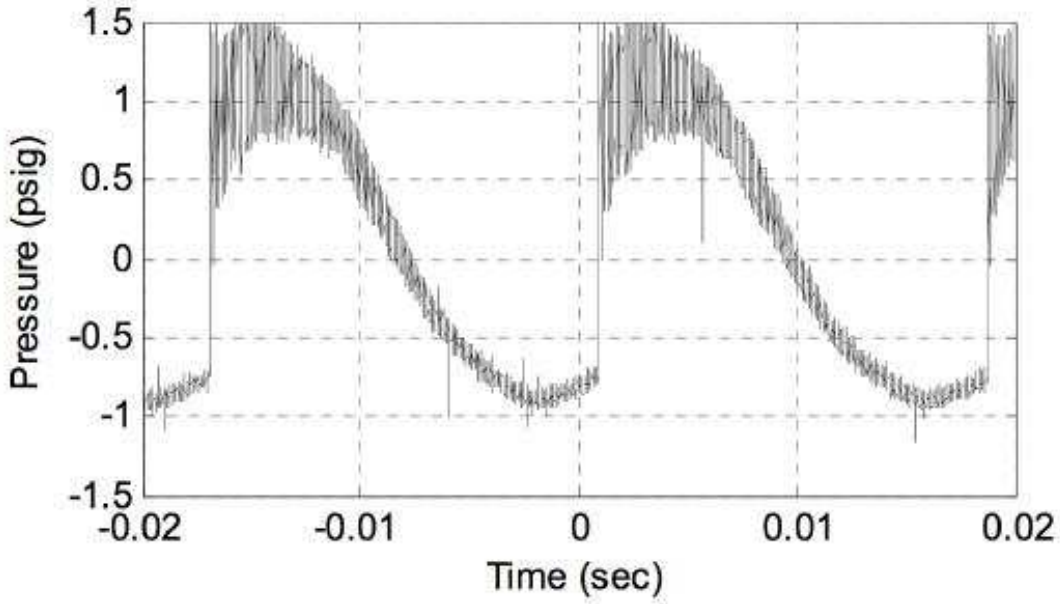


Figure 24: Measured pressure profile at the end of cylinder. The measurement is done after reaching to stationary flow so that the transient increase in pressure profile is not shown here.

value which is balanced with **numerical dissipation**. Therefore the solution doesn't have any physical meaning for attenuation of acoustic waves. We are only interested to see the formation of shock and other characteristics will be investigated in the extension to this work where viscous terms with high-order of accuracy will be added. As shown in (25) the formation of shock is captured even for current solver which is first-order. The results seems to be promising.

3.0 High-Order Discretization in Space Using Least-Square Gradient Reconstruction

An excellent and comprehensive description of least square gradient reconstruction method with application to node-centered grid discretization is given in Ref[2]. Here we only bring

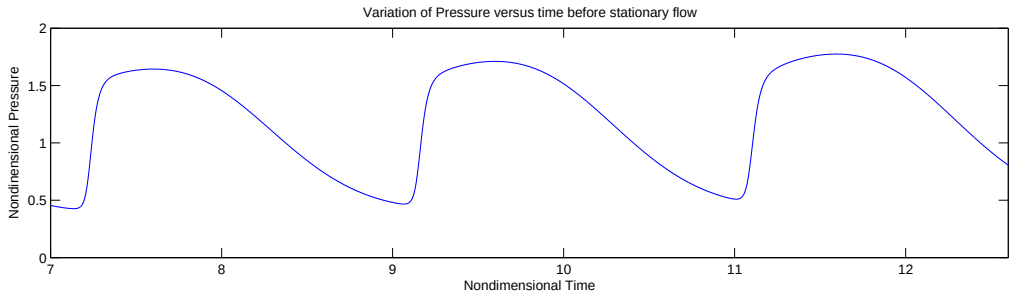


Figure 25: First order in time and space computation of pressure profile at the end of cylinder. The preliminary results show the process of nonlinear pumping and formation of nonlinear waves inside the tube. The results should not be accurate because the grid is coarse and first order is used for both space and time.

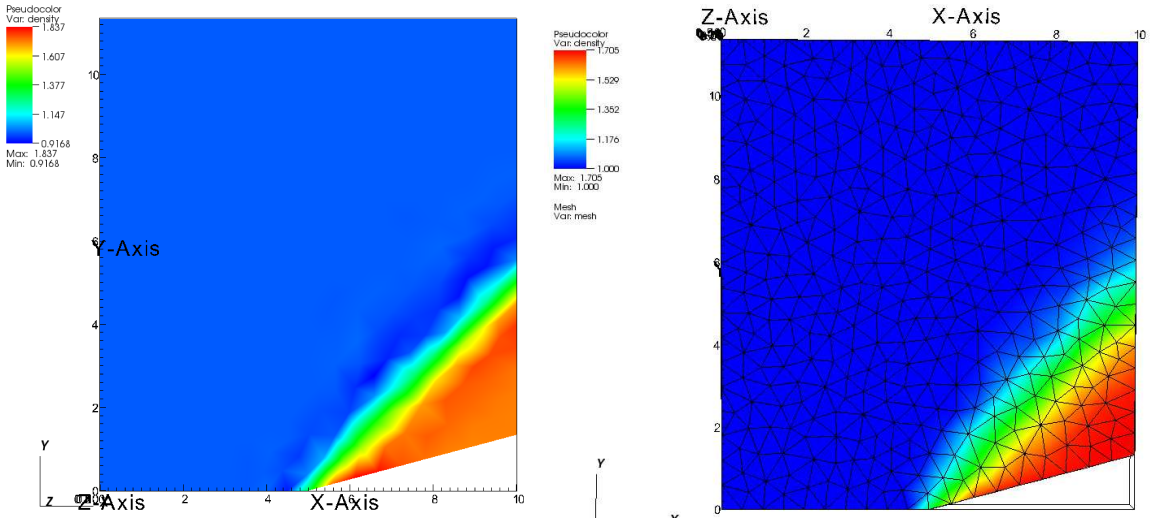


Figure 26: The density contours of solutions obtained using (LEFT) second-order least square flux reconstruction, (Right) Original first-order method.

the result of implementation. After finding the weights per edges, we store them (six weight per edge) to increase the speed of the main loop while sacrificing the memory. Doing so we reconstruct the fluxes near edges using high-order reconstruction for conservative variables. Finally the ramp coarse problem is solved with second-order method and the result are brought in fig.(26).

As shown, the shock is captured better compared to the first-order method. The increase in the spatial resolution of the method is visible. The residual convergence curve for the case of second-order discretization is brought in fig.(27).

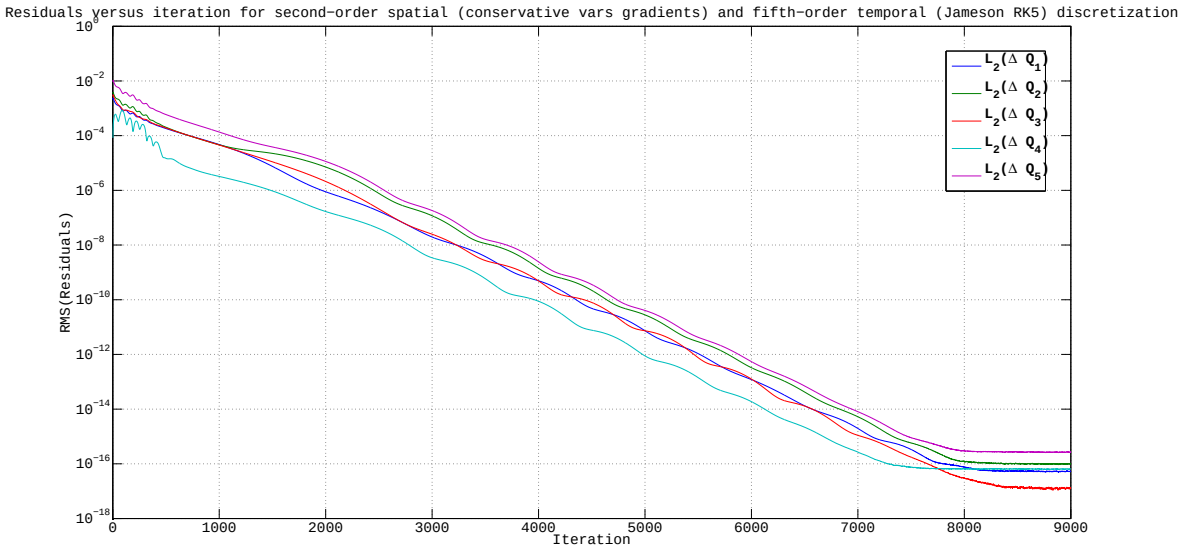


Figure 27: The residuals curve for fig.(26)-LEFT.

4.0 Extending the algorithm to viscous flow fields

Viscous terms are added by first taking the averages of already computed gradients over each edges. Then the shear stress tensor and temperature gradients are evaluated at the center of each edges. After that, instead of changing the code to account for viscous fluxes per area, we add the twice of viscous fluxes per edge to each node so that in the Roe scheme the averaging formula *cancels* our changes. In addition, we changed the ghost-edge based boundary conditions to account for viscous boundary specifications because in this case we need more boundary conditions. To do this the residuals are imposed for known physical quantities (like velocity vector at walls or temperature/temperature gradients at walls and we used characteristic methods for unknown physical variables like density at walls. Finally we validated the code for the famous flat plate benchmark problem. We first created a grid using GridGen commercial software as shown in fig.(28).

The numerical solution is performed with various CFL numbers. Here CFL = 1.1. As the code runs, the momentum at boundary layer penetrates into inviscid flow until it converges

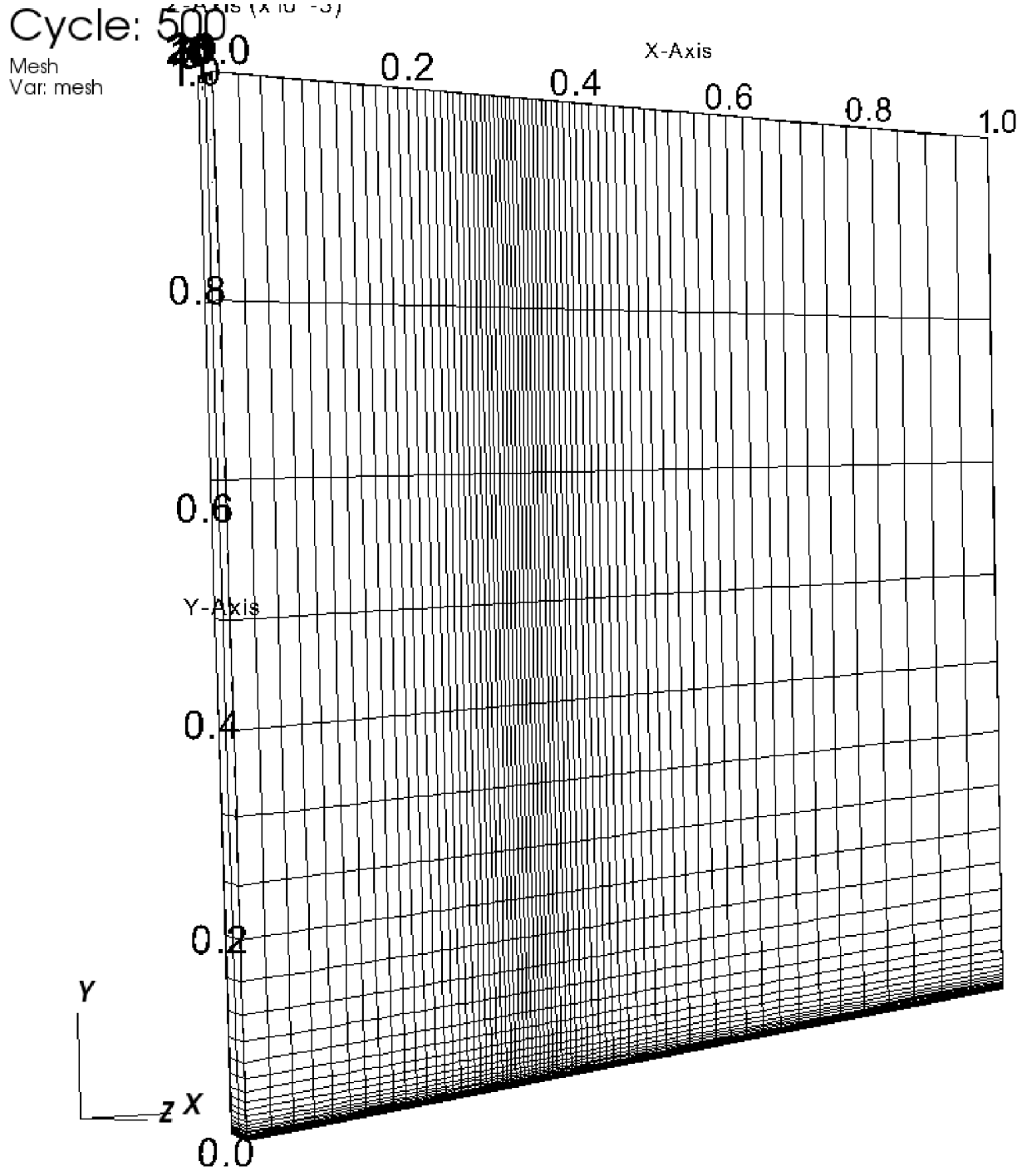


Figure 28: Mesh used for viscous flat plate boundary layer solution.

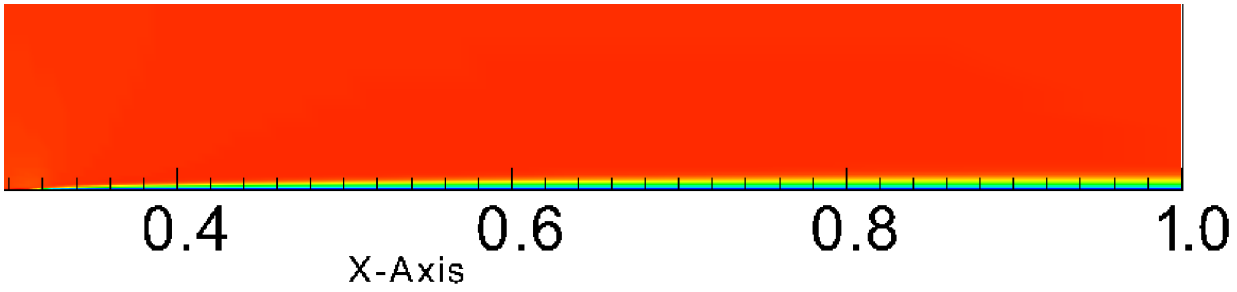


Figure 29: u velocity contours near the boundary region.

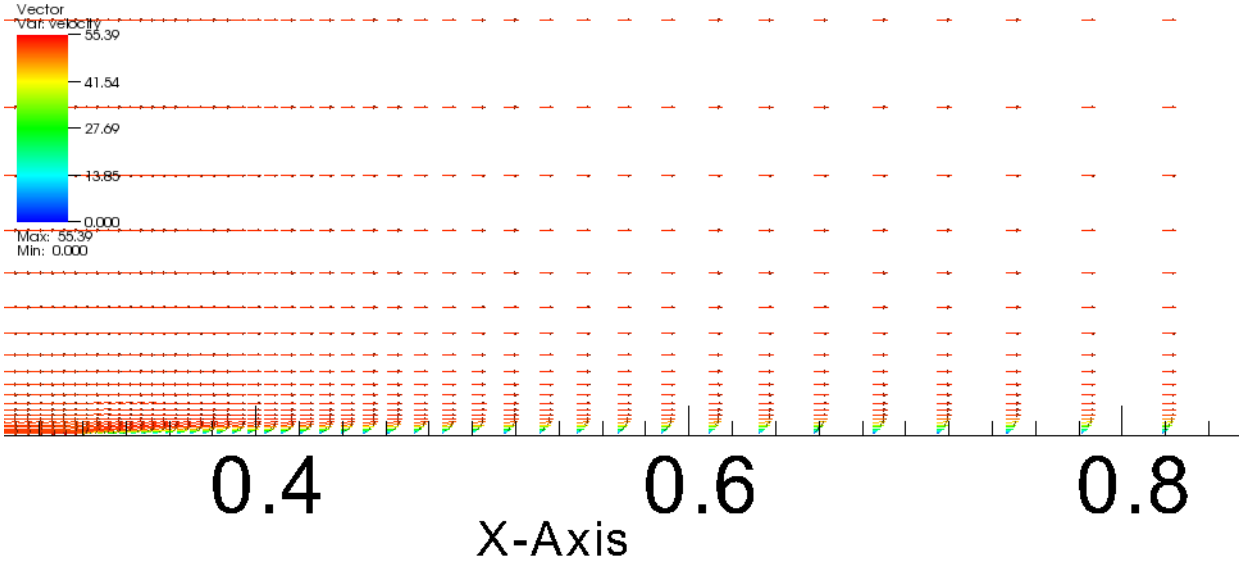


Figure 30: Velocity vectors in the boundary layer and outside of it.

to a steady-state pattern. The final solution for u velocity is shown in fig.(29) and (30).

To compare the result with analytical Blasius solution, we solved the Blasius ODE using a simple shooting method scheme and the code is brought in Appendix B. In addition we used commercial software Fluent to solve the exactly same problem on exactly same grid. An implicit compressible full Navier stokes solver is selected in Fluent. Boundary conditions are imposed in the same manner as we did in our code. The result of comparison is brought in fig.(31). As shown the solution obtained with numerical method in this code is almost like the exact solution while there is a slight deviation in the Fluent's results.

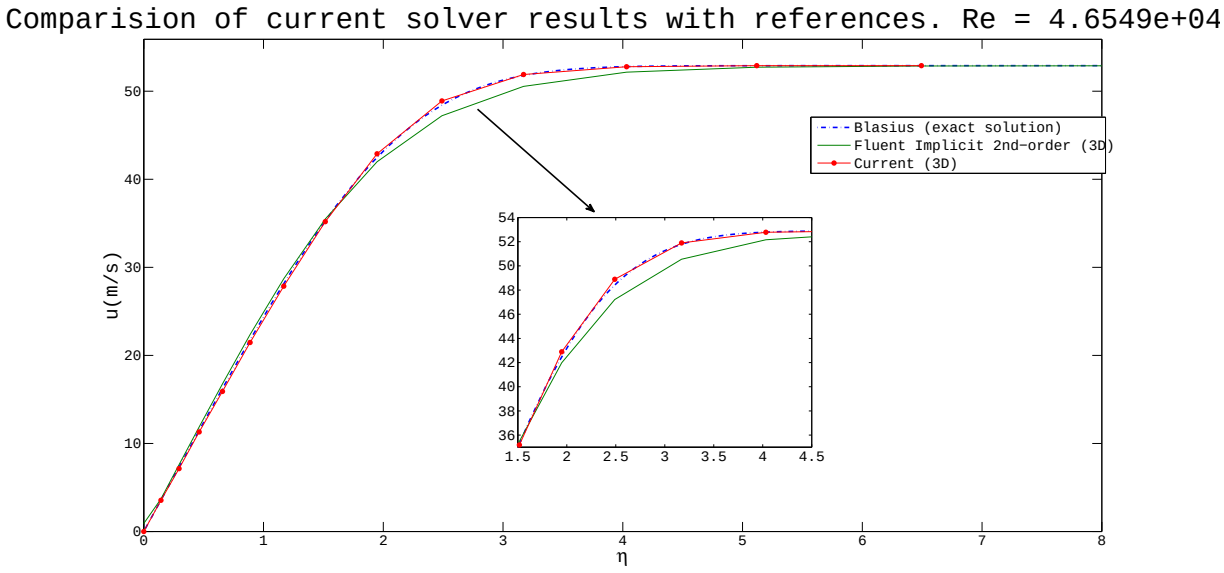


Figure 31: Comparison of exact boundary layer solution with a) Numerical method used in this report. b) Fluent code.

5.0 Conclusions

In this work we started with a built-in grid-reader function and ended with a three-dimensional compressible Navier-Stokes solver for mixed-element unstructured grids. Before writing the solver, we worked on the grid part to create all maps that is needed during the main loop of the solver. In the solver development, we used the Roe scheme for flux differencing and explicit and Runge-Kutta methods for marching in the time. The general purpose of the current work is to give the student confidence to develop their own codes and/or implement their own ideas which is vital in the field of computational engineering.

References

[1] "Practicum Summer Course Lecture Notes", Department of Computational Engineering, University of Tennessee at Chattanooga, USA (Summer 2010)

[2] Daniel G. Hyams, "AN INVESTIGATION OF PARALLEL IMPLICIT SOLUTION ALGORITHMS FOR INCOMPRESSIBLE FLOWS ON UNSTRUCTURED TOPOLOGIES", PHD dissertation,Mississippi State university,May 2000

[3] Anderson, J. D., "Fundamentals of Aerodynamics", McGraw-Hill International Editions

[4] Hirsch, J. D., "Numerical Computation of Internal and External Flows", Butterworth-Heinemann.

[5] Roe, P. L., "Approximate Riemann solvers, parameter vectors, and difference schemes", Journal of Computational Physics, Volume 43, Issue 2, October 1981, Pages 357-372.

Appendix A — Complete derivation of Eigen-Values and Eigen vectors for one-dimensional Euler equations. (Shubin Nozzle)

Appendix B — Blasius boundary layer ODE solver.

This can be easily implemented in Matlab and results converges in a half of a minuet. First write the following ODE function in a seperate file.

```
function out = BlasiusFn(t,y)
out = zeros(3,1);
out(1) = -y(1) * y(3);
out(2) = y(1);
out(3) = y(2);
```

Then run the following script in Matlab and it will gives everything wanted.

```
%blasius solver ver 1.0

clear all;
clc

init = 1.;
error = 1.0;
tole = 1e-14;
t = [0 100];

while (abs(error) >= tole)
[t,u] = ode45(@BlasiusFn,t,[init 0. 0.]);
error = -u(end,2) + 1.0
init = init + error;
end
```

```
%Plotting the profile
figure(1);
plot(t,u(:,1),t,u(:,2),t,u(:,3));
legend('f^{(2)}','f^{(1)}','f');
axis([0. 8. 0. 4.]);
xlabel('y');
title('Blasius exact solution');

%Calculating the specific problem.
U = 52.9000;
rho = 1.205;
x = 0.858385-0.326531;
mu = 1.8208e-5*40.0;
nu = mu/rho;
Reynolds = U*x/nu;
tt = t;
uu = U*u(:,2);
%eta = tt*sqrt(U/(2*nu*x));
%u_blas = U * spline(tt,uu,eta);

%Data from the solver
%y_grid = [0., 0.0005, 0.00102813 0.00161409 0.00229084 0.00309647 0.00407627 0.00528538

u_fluent = [0 0.911903;
0.0005 3.69652;
```

```
0.00102813 7.59084;
0.00161409 11.882;
0.00229084 16.7588;
0.00309647 22.3643;
0.00407627 28.7047;
0.00528538 35.4882;
0.0067918 41.9853;
0.00868028 47.2137;
0.011057 50.5502;
0.0140558 52.1621;
0.0178452 52.7289;
0.0226385 52.8682;
0.0287052 52.8901;
0.0363866 52.8899;
0.0461149 52.8859;
0.0584373 52.8803;
0.0740469 52.8727;
0.0938218 52.8624;
0.118874 52.8487;
0.150614 52.8304;
0.190826 52.8065;
0.241772 52.7766;
0.306319 52.7413;
0.388097 52.7029;
0.491706 52.6648;
0.622975 52.6314];
```

```
y_grid = u_fluent(:,1);
u_fluent = u_fluent(:,2);
eta_grid = y_grid*sqrt(U/(2*nu*x));

%my solution
u_mine=[0;3.54541;7.14249;11.3066;15.9112;21.4696;27.8553;35.1873;42.8937;48.8933;51.898
figure(2);

plot(tt,uu,'-.',eta_grid,u_fluent,'-',eta_grid(1:length(u_mine)),u_mine,'-');
legend('Blasius (exact solution)', 'Fluent Implicit 2nd-order (3D)','Current (3D)');
axis([0 8 0 U+3]);
title('Comparision of current solver results with references. Re = 4.6549e+04.');
%Current implementation include viscous full compressible 3D NS solver on
%general unstructured grid
xlabel('y(m)');
ylabel('u(m/s)');
```

The solution is brought in fig.(32).

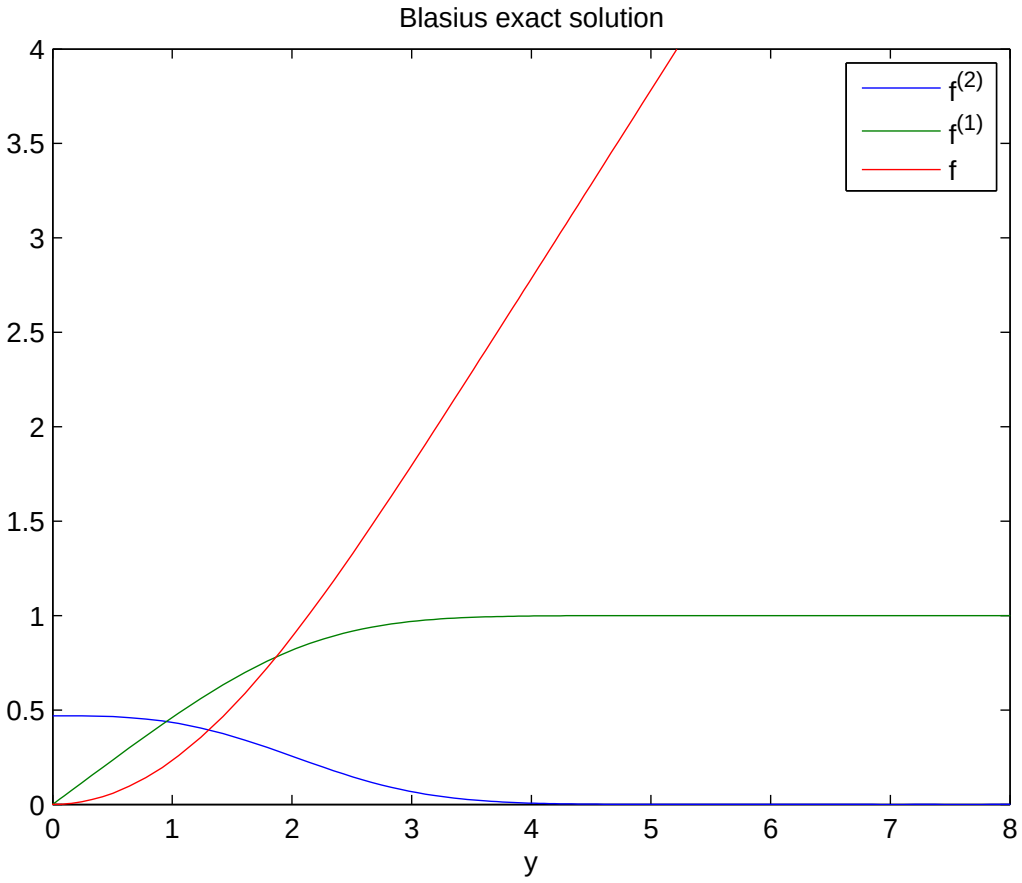


Figure 32: The exact Blasius solution.