



**ODTÜ METU**

**ME485: Computational Fluid Dynamics  
Using Finite Volume Method**

Homework 3

**Uğur Ozan Baygeldi**

**Ali Kaya**

**Onur Kakilli**

Group 15

## 1 Introduction

This report is on the second homework of the course ME485, Computational Fluid Dynamics Using Finite Volume Method. The homework requests integration of limiters and upwinding methods to *mefvm*.

## 2 Roadmap

This homework has an extra step to implement unlike other two completed beforehand. The extra task is to implement a feature called "Barth-Jespersen limiter" which is generally used in highly turbulent, supersonic, or even hypersonic applications to record flow or shock properties in high definitions. The Homework requires to implement:

- Implement `_make_compute_norm`, `_make_compute_fpts` and `_make_barth_jespersen` to `AdvectionElements` class at `elements.py`
- Implement `make_minmax` to `AdvectionIntInter` and `AdvectionBCInter` classes at `inters.py`
- Implement `make_flux` to `AdvectionIntInter` and `AdvectionBCInter` classes at `inters.py`
- Implement `upwind` and `Local-Lax-Friedrich` (Rusanov) flux functions to `AdvectionIntInter` at `fsolver.py`

Following the same methodology used in the previous two homework the roadmap was implemented according to the order called at `system.py`. Using this method also allows for checking memory errors by printing the word here occasionally after certain steps.

Furthermore, this homework requires checking different conditions over different meshes which is done by implementing different equations and creating different meshes using `gmsh`.

### 3 Implementation of `_make_compute_fpts` and `_make_barth_jespersen` at `elements.py`

#### 3.1 Implementation of `_make_compute_fpts`

The implementation of `_make_compute_fpts` is done easily after numerous implementations in homeworks 1 and 2. The code simply assigns the cell center values of elements for the given function to faces.

To implement this function the following code is utilized:

```
1 def _make_compute_fpts(self):
2     nvars, nface = self.nvars, self.nface
3     def _compute_fpts(i_begin, i_end, upts, fpts):
4         # Code taken directly from HW1
5         for idx in range(i_begin, i_end):
6             for j in range(nvars):
7                 for i in range(nface):
8                     fpts[i, j, idx] = upts[j, idx]
9     return self.be.make_loop(self.neles, _compute_fpts)
```

#### 3.2 Implementation of `_make_barth_jespersen`

Barth-Jespersen limiter is a commonly used limiter in the world of CFD. A limiter is used to prevent non-physical results rising from oscillations from appearing in the simulations.

##### 3.2.1 Mathematical Implementation of `_make_barth_jespersen`

The Barth-Jespersen limiter is simply the following. (Karakuş 2024b)

$$\Psi_0 = \min_f \begin{cases} \min \left( 1, \frac{\phi_{\max} - \phi_0}{\Delta_f} \right) & \text{if } \Delta_f > 0, \\ \min \left( 1, \frac{\phi_{\min} - \phi_0}{\Delta_f} \right) & \text{if } \Delta_f < 0, \\ 1 & \text{if } \Delta_f = 0, \end{cases} \quad (1)$$

Where:

- $\phi_{\max}$  and  $\phi_{\min}$  are the maximum and minimum values of  $\phi$  in the stencil.
- $\phi_0$  is the current cell value.
- $\Delta_f = \nabla q_0 \cdot \mathbf{r}_{0,f}$  is the change in the unlimited reconstructed values at the face.

### 3.2.2 Code Implementation of `_make_barth_jespersen`

The Barth-Jespersen limiter is implemented in the code using the following statements:

```

1  def _make_barth_jespersen(self, limiter):
2      nface, ndims, nvars = self.nface, self.ndims, self.nvars
3      dxf = self.dxf
4      # Compiler arguments
5      array = self.be.local_array()
6
7      def _cal_barth_jespersen(i_begin, i_end, upts, grad, fext, lim):
8          for i in range(i_begin, i_end):
9              for j in range(nvars):
10                 fiList = np.zeros(nface)
11                 fiMax = max(fext[0, :, j, i])
12                 fiMin = min(fext[1, :, j, i])
13                 for f in range(nface):
14                     deltaF = dot(grad[:, j, i], dxf[f, :, i], ndims) + eps
15                     if deltaF > 0:
16                         fiList[f] = min(1, (fiMax - upts[j, i])/deltaF)
17                     elif deltaF < 0:
18                         fiList[f] = min(1, (fiMin - upts[j, i])/deltaF)
19                     elif deltaF == 0:
20                         fiList[f] = 1
21                 lim[j, i] = min(fiList)
22      return self.be.make_loop(self.neles, _cal_barth_jespersen)

```

The code implements what was stated in the mathematical expressions. First checks the delta F value and then determines the limits.

After the limits are determined, the `lim` array is filled appropriately to be used further in the flux calculations.

## 4 Implementation of `_make_minmax`

The function `_make_minmax` is used to determine the maximum and minimum of neighboring cell faces. The function then simply assigns the max and min to the specified position in the `fext` array.

### 4.1 Code Implementation of `_make_minmax`

The following code implements the method. Note that for the below code the array name is `uext`, which will be used as the `fext` array mentioned in *chapter 4*

```
1  def _make_minmax(self):
2      nvars = self.nvars
3      lt, le, lf = self._lidx
4      rt, re, rf = self._ridx
5
6      def compute_minmax(i_begin, i_end, uf, *uext):
7          for idx in range(i_begin, i_end):
8              lti, lfi, lei = lt[idx], lf[idx], le[idx]
9              rti, rfi, rei = rt[idx], rf[idx], re[idx]
10
11             for j in range(nvars):
12                 ul = uf[lti][lfi, j, lei]
13                 ur = uf[rti][rfi, j, rei]
14
15                 uext[lti][0, lfi, j, lei] = max(ul, ur)
16                 uext[lti][1, lfi, j, lei] = min(ul, ur)
17
18                 uext[rti][0, rfi, j, rei] = max(ul, ur)
19                 uext[rti][1, rfi, j, rei] = min(ul, ur)
20
21             return self.be.make_loop(self.nfpts, compute_minmax)
```

Similarly, the `bc` function is required to adjust the "ghost elements" neighboring the boundary element for the `AdvectionBCInters` class.

However, for the boundary case, since there is no right element, there is no setting of the right element position at `uext`.

The code for the boundary elements are as follows:

```
1  def _make_minmax(self):
2      nvars = self.nvars
3      ndims = self.ndims
4      lt, le, lf = self._lidx
5      nf = self._vec_snorm
6
7      bc = self.bc
8      array = self.be.local_array()
9
10     def compute_minmax(i_begin, i_end, uf, *uext):
11
12         for idx in range(i_begin, i_end):
13             lti, lfi, lei = lt[idx], lf[idx], le[idx]
14             ur = array(nvars)
15             vr = array(ndims)
16             vl = array(ndims)
17
18             ul = uf[lti][lfi, :, lei]
19             bc(ul, ur, vl, vr, nf[:, idx])
20
21             for j in range(nvars):
22
23                 uext[lti][0, lfi, j, lei] = max(ul[j], ur[j])
24                 uext[lti][1, lfi, j, lei] = min(ul[j], ur[j])
25
26     return self.be.make_loop(self.nfpts, compute_minmax)
```

## 5 Implementation of `_make_flux`

The purpose of setting limiters and calculating gradients is to obtain the flux values. This section of the code ties everything together yielding a sensible result.

### 5.1 Code Implementation of `_make_flux`

To implement `_make_flux` for the `AdvectionIntInters` the following code is utilized. This function is similar to a case for the actual flux schemes.

A *similar code* will be used for the `AdvectionBCInters` which will be explained later:

```

1  def _make_flux(self):
2      ndims, nfvars = self.ndims, self.nfvars
3      lt, le, lf = self._lidx
4      rt, re, rf = self._ridx
5      nf, sf = self._vec_snorm, self._mag_snorm
6      # Compiler arguments
7      array = self.be.local_array()
8      cplargs = {
9          'flux_func' : self.ele0.flux_container(),
10         'ndims' : ndims,
11         'nfvars' : nfvars,
12         'array' : array,
13         **self._const
14     }
15
16     # Get numerical schems from `rsolvers.py`
17     scheme = self.cfg.get('solver', 'flux')
18     flux = get_fsolver(scheme, self.be, cplargs)
19
20     def comm_flux(i_begin, i_end, vf, *uf):
21         for idx in range(i_begin, i_end):
22             # flux function to be filled
23             fn = array(nfvars)
24             lti, lfi, lei = lt[idx], lf[idx], le[idx]
25             rti, rfi, rei = rt[idx], rf[idx], re[idx]
26             nfi = nf[:, idx]
27
28             ul = uf[lti][lfi, :, lei]
29             ur = uf[rti][rfi, :, rei]
30
31             vl = vf[lti][lfi, :, lei]
32             vr = vf[rti][rfi, :, rei]
33
34             # call the numerical flux function here : i.e. upwind or rusanov
35             flux(ul, ur, vl, vr, nfi, fn)
36
37         for jdx in range(nfvars):
38             # Save it at left and right solution array
39             uf[lti][lfi, jdx, lei] = fn[jdx]*sf[idx]
40             uf[rti][rfi, jdx, rei] = -fn[jdx]*sf[idx]
41
42     return self.be.make_loop(self.nfpts, comm_flux)

```

$m_{eff}^{fm}$  computes the different flux schemes separately in a file called `fsolvers.py` which will be further explained in *chapter 6*. The code takes the required arguments and feeds them to the flux function. Which also has the scheme info. (lines 17 & 18)

For the AdvectionBCInters, similar to `_make_minmax` the `bc` function is called.

```
1  def _make_flux(self, nele):
2      ndims, nfvars = self.ndims, self.nfvars
3      lt, le, lf = self._lidx
4      nf, sf = self._vec_snorm, self._mag_snorm
5
6      # Compiler arguments
7      array = self.be.local_array()
8      cplargs = {
9          'flux_func' : self.ele0.flux_container(),
10         'ndims' : ndims,
11         'nfvars' : nfvars,
12         'array' : array,
13         **self._const
14     }
15
16     # Get numerical schems from `rsolvers.py`
17     scheme = self.cfg.get('solver', 'flux')
18     flux = get_fsolver(scheme, self.be, cplargs)
19     # Get bc function
20     bc = self.bc
21
22     def comm_flux(i_begin, i_end, vf, *uf):
23
24         for idx in range(i_begin, i_end):
25             fn = array(nfvars)
26
27             lti, lfi, lei = lt[idx], lf[idx], le[idx]
28             nfi = nf[:, idx]
29             ul = uf[lti][lfi, :, lei]
30             vl = vf[lti][lfi, :, lei]
31             vr = array(ndims)
32             ur = array(nfvars)
33             bc(ul, ur, vl, vr, nfi)
34
35             flux(ul, ur, vl, vr, nfi, fn)
36
37             for jdx in range(nfvars):
38                 # Save it at left solution array
39                 uf[lti][lfi, jdx, lei] = fn[jdx]*sf[idx]
40
41     return self.be.make_loop(self.nfpts, comm_flux)
```



## 6 Implementation of Upwind and Local-Lax-Friedrich (Rusanov)

Upwind and Local-Lax-Friedrich (Rusanov) are both flux schemes used commonly. Both of these schemes have strong sides and weak sides.

### 6.1 Mathematical Implementation of Upwind

The upwind flux determines the flux based on the direction of the velocity field:

$$F_{\text{upwind}} = \begin{cases} \mathbf{v}\phi_{\text{upwind}} & \text{if } \mathbf{v} \cdot \mathbf{n} > 0, \\ \mathbf{v}\phi_{\text{downwind}} & \text{if } \mathbf{v} \cdot \mathbf{n} \leq 0, \end{cases} \quad (2)$$

- $\phi_{\text{upwind}}$  is the value of  $\phi$  in the upstream direction (where the flow originates).
- $\phi_{\text{downwind}}$  is the value of  $\phi$  in the downstream direction (where the flow exits).

This method ensures stability by using information from the direction of incoming flow.

### 6.2 Code Implementation of Upwind

The main implementation of the flux schemes relies on the `flux_func` argument taken from the `advection.ini` file. Where the code parses the equation to be machine-readable.

The code to implement Upwind relies on the dot product of vectors to determine the flow direction after that the flux at the interface can be selected which can be simply implemented using the following block:

```
1 def make_upwind(cplargs):
2     nvars      = cplargs['nvars']
3     gamma      = cplargs['gamma']
4     flux_func  = cplargs['flux_func']
5     array      = cplargs['array']
6     ndims      = cplargs['ndims']
7     def fsolver(ul, ur, vl, vr, nf, fn):
8
9         fl = array(nvars)
10        fr = array(nvars)
```

```

11     flux_func(ul, vl, nf, fl)
12     flux_func(ur, vr, nf, fr)
13
14     if dot(vl, nf, ndims) > 0:
15         for i in range(nvars):
16             fn[i] = fl[i]
17     else:
18         for i in range(nvars):
19             fn[i] = fr[i]
20
21     return fsolver

```

### 6.3 Mathematical Implementation of Local-Lax-Friedrich (Rusanov) flux

The Local Lax-Friedrichs flux, also known as the Rusanov flux, provides a balance between stability and accuracy by incorporating both the central flux and a dissipation term. It is expressed as:

$$F_{\text{Rusanov}} = \frac{1}{2} [\mathbf{v}(\phi_{\text{left}} + \phi_{\text{right}}) - |\lambda|(\phi_{\text{right}} - \phi_{\text{left}})] \quad (3)$$

Here:

- $\phi_{\text{left}}$  and  $\phi_{\text{right}}$  are the values of  $\phi$  on the left and right sides of the interface, respectively. These represent the reconstructed values from adjacent cells.
- $|\lambda|$  is the spectral radius (or the maximum eigenvalue) of the Jacobian matrix of the flux function. In the case of advection,  $|\lambda|$  corresponds to the maximum magnitude of the velocity,  $|\mathbf{v}|$ , across the interface.

The Rusanov flux can be understood in two parts:

1. **Central Flux:** The term  $\mathbf{v}(\phi_{\text{left}} + \phi_{\text{right}})$  computes the average flux contribution from both sides of the interface. This ensures a symmetric treatment of the interface.
2. **Dissipation Term:** The term  $-|\lambda|(\phi_{\text{right}} - \phi_{\text{left}})$  introduces numerical dissipation, which stabilizes the solution by dampening oscillations or discontinuities at the interface.

The balance between these two components allows the Rusanov flux to handle sharp gradients and discontinuities in  $\phi$  while maintaining numerical stability. This makes it particularly effective for solving hyperbolic equations such as the advection equation.

## 6.4 Code Implementation of Local-Lax-Friedrich (Rusanov) flux

To implement Local-Lax-Friedrich (Rusanov) flux the following code is used.

```

1 def make_rusanov(cplargs):
2     nvars      = cplargs['nvars']
3     gamma      = cplargs['gamma']
4     flux_func  = cplargs['flux_func']
5     array      = cplargs['array']
6     ndims      = cplargs['ndims']
7     def fsolver(ul, ur, vl, vr, nf, fn):
8         fl = array(nvars)
9         fr = array(nvars)
10
11         # this is u*phi*n
12         flux_func(ul, vl, nf, fl)
13         flux_func(ur, vr, nf, fr)
14         a = max(abs(dot(vl, nf, ndims)), abs(dot(vr, nf, ndims)))
15
16         for i in range(nvars):
17             fn[i] = 0.5 * (fl[i] + fr[i]) - 0.5 * a * (ur[i] - ul[i])
18
19     return fsolver

```

The  $\lambda$  value mentioned in *section 6.3* is calculated by comparing absolute values of the velocities of the elements normal to the face and picking the maximum. After which the Rusanov scheme is used.

## 7 Implementation of `_make_compute_norm`

This homework requires the area to be calculated which will be taken as the norm. To actually calculate the norm first calculated area is subtracted from the last area.

## 7.1 Mathematical Implementation of `_make_compute_norm`

To implement `_make_compute_norm` for this homework, there is no need for the exact solution. Essentially, the sum of all the element volumes with  $\phi \leq 0$ .

## 7.2 Code Implementation of `_make_compute_norm`

The code to implement the requested norm uses the array `upts`, note that the index 0 of the array is used. This index houses the  $\phi$  values. With an if statement the element is checked to see if it satisfies the condition. If it does the volume of the element is added to the total.

The code to implement `_make_compute_norm` is as follows:

```
1  def _make_compute_norm(self):
2      import numba as nb
3      vol = self._vol
4      neles, nvars, ndims = self.neles, self.nvars, self.ndims
5      xc = self.xc.T
6      def run(upts):
7          norm = np.zeros(nvars)
8          for i in range(neles):
9              for j in range(nvars):
10                 if upts[0][j, i] <= 0:
11                     norm[j] += vol[i]
12             return norm
13
14     return self.be.compile(run, outer=True)
```

## 8 Tests and Results

In this part of the report, tests of various geometries with different mesh structures (triangular/quadrilateral, structured/unstructured); different test cases by changing level set functions and velocity profiles; different numeric flux calculation methods, and lastly with varying steps of time will be conducted to see the boundaries of the code implemented.

All the results will be tabulated for easier viewing and commenting. Furthermore, you can visit [here](#) to see the output.

## 8.1 Meshes

A table explaining all the meshes, with node and element counts, will be provided in the *appendix*

The first mesh tested is already provided within the homework code (Karakuş 2024a). As seen from the figure below, it consists of a rectangle divided into two equal parts. Although they are symmetric, each side has a different mesh type. While one side contains quadrilateral mesh elements, the other contains triangular-type elements. Moreover, the quadrilateral side is also more structured than the triangle side. This creates a test for the interface between quadrilateral and triangular mesh types, allowing to check for any numerical errors.

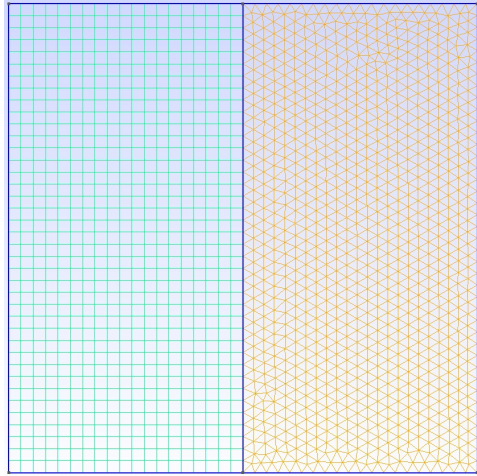


Figure 1: Provided Test Mesh Geometry

The second test mesh is constructed to see the effect of "division" number of main geometry. It is also seasoned with two structural quadrilateral elements and two unstructured triangular elements. In this mesh, more transitions between the element shapes are observed. This mesh has 3 interface sides with different element types which is 2 more than the first mesh.

The mesh can be seen on the next page.

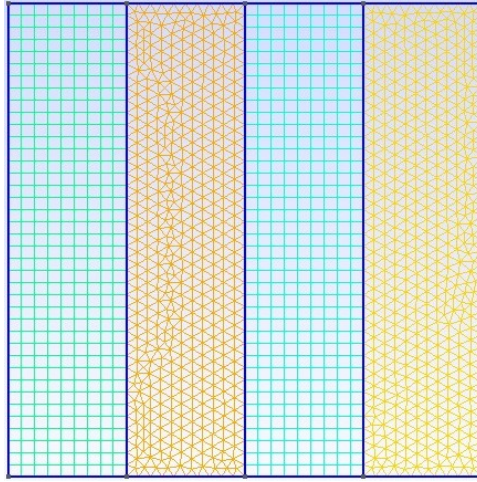


Figure 2: Four Section Structured/Unstructured Test Mesh Geometry

The third mesh is a purely triangular mesh which will be used along the fourth mesh to check whether quadrilateral or triangular elements are better. These two meshes will also help compare the different velocity fields.

The third mesh is as follows:

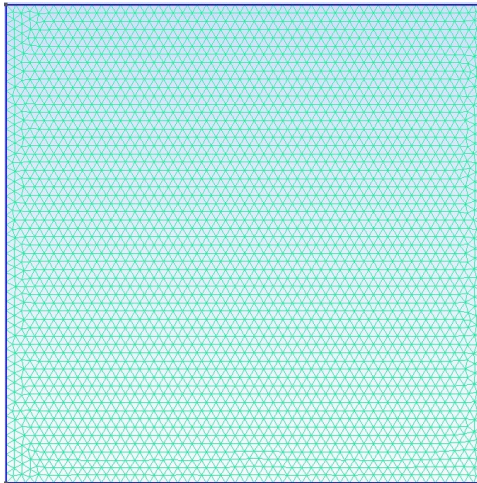


Figure 3: Triangular Element Test Mesh Geometry

Similarly, the fourth mesh is a fully quadrilateral mesh with relative fineness to the triangular one.

The fourth mesh can be seen on the next page.

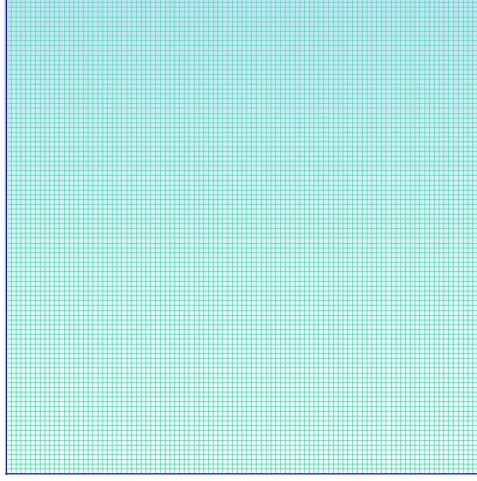


Figure 4: Quadrilateral Element Test Mesh Geometry

## 8.2 Velocity Fields

To test the meshes, several different velocity fields have been created.

The first field is given by the instructor (Karakuş 2024a). Which creates a spiraling velocity field that does not mix the variable. This field can be expressed with the following expression:

$$u_x = -2\pi(y - 0.5) \text{ \& } u_y = 2\pi(x - 0.5)$$

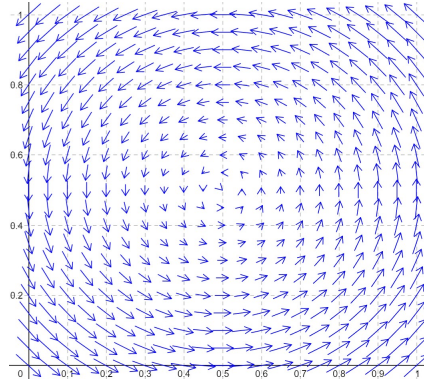


Figure 5: Provided Velocity Field (Field 1)

The second field is created to check the mixing of the field. This is done using the following expression:

$$u_x = \sin(\pi x)\cos(\pi y) \text{ \& } u_y = -\cos(\pi x)\sin(\pi y)$$



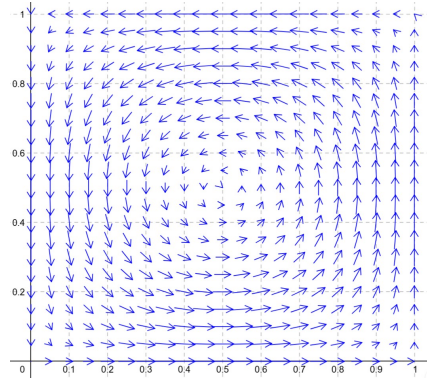


Figure 6: Mixing Velocity Field (Field 2)

The third field is pure advection towards the right. The velocity field is:

$$u_x = 0.3 \text{ \& } u_y = 0.0$$

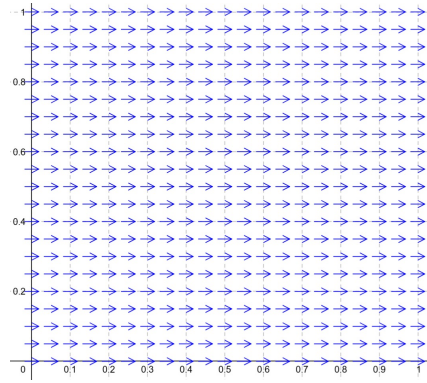


Figure 7: Advection to Right Velocity Field (Field 3)

The fourth velocity field has a discontinuity which prevents the calculations hence it will not be used. The field can be expressed as:

$$u_x = y - 0.5 \text{ \& } u_y = 0.0$$

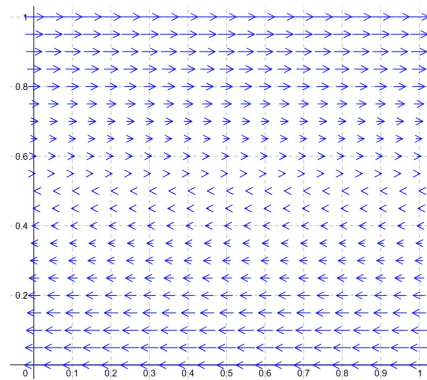


Figure 8: Discontinuous Velocity Field (Field 4)



These velocity fields are, again, tabulated at the *appendix*.

### 8.3 Obtained Norms

The norms obtained for the different schemes using different meshes are as follows, refer to the caption to determine the functions.

#### 8.3.1 Timestep = 0.33





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0018640]	[0.0018640]
 Figure 2	[0,0073474]	[0.0073479]
 Figure 3	[0.0]	[0.0]
 Figure 4	[0.0001020]	[0.0001020]

Table 1: Results for Given Function with Field 1 Dirichlet





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[-0.0015359]	[-0.0015359]
 Figure 2	[0.0195493]	[0.0195493]
 Figure 3	[-0.0005196]	[-0.0005196]
 Figure 4	[0.0]	[0.0]

Table 2: Results for Given Function with Field 2 Dirichlet





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0092314]	[0.0092314]
 Figure 2	[0.0212781]	[0.0212781]
 Figure 3	[0.0000127]	[0.0000127]
 Figure 4	[0.0007142]	[0.0007142]

Table 3: Results for Given Function with Field 3 Dirichlet

### 8.3.2 Timestep = 0.66





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0054839]	[0.0054839]
 Figure 2	[0.0073479]	[0.0697943]
 Figure 3	[0.0]	[0.0]
 Figure 4	[0.0]	[0.0]

Table 4: Results for Given Function with Field 1 Dirichlet





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[-0.0015359]	[-0.0015359]
 Figure 2	[0.0195493]	[0.0195493]
 Figure 3	[-0.0005196]	[-0.0005196]
 Figure 4	[0.0]	[0.0]

Table 5: Results for Given Function with Field 2 Dirichlet





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0060644]	[0.0060644]
 Figure 2	[0.0309567]	[0.0309567]
 Figure 3	[0.0000127]	[0.0000127]
 Figure 4	[0.0008162]	[0.0008162]

Table 6: Results for Given Function with Field 3 Dirichlet

### 8.3.3 Timestep = 1.0





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0018640]	[0.0018640]
 Figure 2	[0.0073479]	[0.0073479]
 Figure 3	[0.0]	[0.0]
 Figure 4	[0.0001020]	[0.0001020]

Table 7: Results for Given Function with Field 1 Dirichlet





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[-0.0055753]	[-0.0055753]
 Figure 2	[0.0222544]	[0.0222544]
 Figure 3	[-0.0006928]	[-0.0006928]
 Figure 4	[0.0]	[0.0]

Table 8: Results for Given Function with Field 2 Dirichlet





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0092314]	[0.0092314]
 Figure 2	[0.0697943]	[0.0212781]
 Figure 3	[0.0001917]	[0.0001917]
 Figure 4	[0.0007142]	[0.0007142]

Table 9: Results for Given Function with Field 3 Dirichlet

### 8.3.4 Modifications to the Function

To modify the function the `r` value and the `yc` values are adjusted in the `advection.ini` file. All the calculations are ran for a time step of 1.0 and the given velocity field (*Field 1*) for simplicity.





Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[-0.0034597]	[-0.0034597]
 Figure 2	[-0.0042726]	[-0.0042726]
 Figure 3	[0.0010392]	[0.0010392]
 Figure 4	[0.0005101]	[0.0005101]

Table 10: Results for  $r = 0.05$  with Field 1 Dirichlet

This value alters the region of influence. One would expect a more present value change between the two types, however, due to the lack of quality of the meshes, the difference is not too noticeable.



Mesh	Norm (Upwind)	Norm (Rusanov)
 Figure 1	[0.0]	[0.0]
 Figure 2	[0.0]	[0.0]
 Figure 3	[0.0]	[0.0]
 Figure 4	[0.0]	[0.0]

Table 11: Results for  $yc = 0.5$  with Field 1 Dirichlet

This value changes the starting location of the influence. For the first two figures due to the interface between the two element types the value changes slightly during operation but as the function rotates due to the purely rotational velocity field the value becomes the same.

#### 8.4 Comments

As a first comment about the tests performed, when the results of the Four Section Structured/Unstructured Test Mesh are compared with the provided test mesh, it is clearly seen that the area norms obtained in the provided test mesh are smaller in every case. Another striking trend is the general decrease in the area norms obtained in the Four Section Structured/Unstructured Test Mesh when the time step is reduced. Unfortunately, however, this trend cannot be observed in the results obtained from the tests performed for the provided mesh. In addition, when the tested meshes are examined separately, the area norms calculated depending on the tested velocity fields change significantly in the Four Section Structured/Unstructured Test Mesh. Although similar changes are observed in the provided mesh, the changes as serious as in the Four Section Structured/Unstructured Test Mesh are not observed.

In addition, when the flux calculation methods were changed from Local-lax Fredric to up-winding scheme, no difference was observed in the calculated area norms.

The next comment can be obtained by comparing the norms of Field 1 and Field 2 in any flux method, and the mesh type has the lowest area norm since these fields do not interfere with the generic structure of initial conditions. Norms obtained are basically numeric diffusion.

Another comment is we first expected to see better results for field 1 when the mesh element is triangular and for field 3 when the element type is quadrilateral since it is expected to see advection compatibility of quadrilateral mesh elements is better than triangular counterparts and vice versa for rotating. Yet, the results indicate the exact opposite of this insight. Triangular mesh elements are more suitable for advection motion while

The next comment is about field 4. As one can see through the report only three tables were provided for each time step. These tables represent each vector field for that time step. Despite having four vector fields, only three of them have results because of the reason shared previously. The velocity discontinuity at the middle causes the code to fail and therefore there are no results for that vector field.

Normally, one would expect a bigger difference between the two flux schemes. However, as the meshes are coarse the difference is not noticeable.

As an added bonus, throughout the code machine epsilon was added to certain equations to prevent division by 0 errors that may arise from truncation and round-off errors.

## References

- Baygeldi-Kaya-Kakilli (2024a). *Homework 1 Report*. Last accessed 05/01/2024. URL: [https://github.com/uobaygeldi/ME485\\_HWs/blob/main/HW1%20Report.pdf](https://github.com/uobaygeldi/ME485_HWs/blob/main/HW1%20Report.pdf).
- (2024b). *Homework 2 Report*. Last accessed 05/01/2024. URL: [https://github.com/uobaygeldi/ME485\\_HWs/blob/main/HW2%20Report.pdf](https://github.com/uobaygeldi/ME485_HWs/blob/main/HW2%20Report.pdf).
- Karakuş, Ali (2024a). *ME485 mefîm HW2 Code*. Last accessed 05/01/2024. URL: <https://github.com/AliKarakus/me485-HWs/tree/HW2>.
- (2024b). *ME485 Lecture Notes*. Last accessed 05/01/2024. URL: <https://odtuclass2024f.metu.edu.tr/course/view.php?id=2835>.

## A Mesh Table

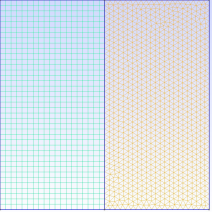
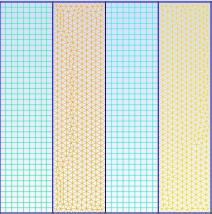
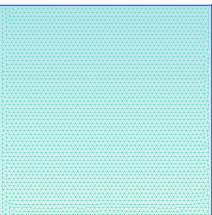
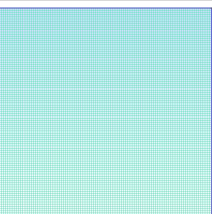
Mesh	Element Count	Node Count	Reason
 <p>Figure 1</p>	2782	1740	Default mesh
 <p>Figure 2</p>	2863	1720	More transition between types
 <p>Figure 3</p>	6028	3013	A fine triangular grid
 <p>Figure 4</p>	10201	10000	A fine structured grid

Table 12: Statistics of Meshes



## B Velocity Fields

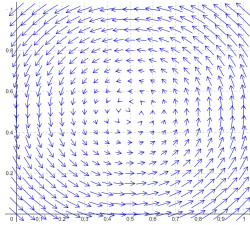
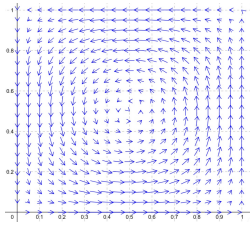
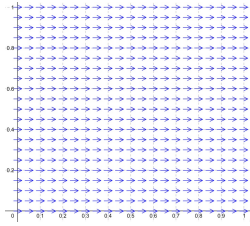
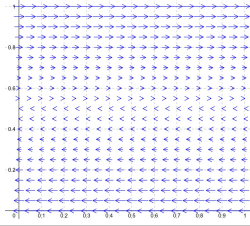
Field	$u_x$	$u_y$	$u_z$
 <p>Field 1</p>	$-2\pi(y - 0.5)$	$2\pi(x - 0.5)$	na
 <p>Field 2</p>	$\sin(\pi x)\cos(\pi y)$	$-\cos(\pi x)\sin(\pi y)$	na
 <p>Field 2</p>	0.3	0.0	na
 <p>Field 2</p>	$y - 0.5$	0.0	na

Table 13: Different Velocity Fields