



ODTÜ METU

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

Homework 2

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 some implementations of diffusion methods to *me_fvm*.

2 Roadmap

This report follows a route mentioned in the previous homework of Group 15, which is initially looking at the `system.py` file and implementing the methods requested as the methods appear on the code. (Baygeldi-Kaya-Kakilli 2024) The purpose of this homework is to complete the missing parts of the code such that the following are implemented successfully:

- Implement `_make_compute_fpts` and `_make_grad` to `ParabolicElements` class at `elements.py`
- Implement `make_grad_at_face` and `_make_flux` to `ParabolicIntInterers` and `ParabolicBCInterers` classes at `inters.py`
- Implement `_make_compute_norm` to `elements.py`

Although there are arguably fewer requests from the previous Homework, a tough and challenging path remains to be solved.

3 Implementation of `_make_grad` and `_make_compute_fpts` at `elements.py`

3.1 Implementation of `_make_compute_fpts`

`_make_compute_fpts` is implemented the same as the previous homework. The code simply assigns cell center values to every face of an element. This is generally the first step in the CFD methods.

The implemented code:

```
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_grad

Similarly, the code is exactly the same with the previous homework. Unlike the previous homework, in the method uses Least Squares to determine the gradient which from the outcomes of the last homework was determined to work better in general.

The code for the operator is, once again, the same used in the previous homework. As implementing the operator is not requested, the code for the operator will not be shared to prevent bulk.

The code:

```
1 def _make_grad(self):
2     nface, ndims, nvars = self.nface, self.ndims, self.nvars
3     # Gradient operator
4     op = self._prelsq
5     def _cal_grad(i_begin, i_end, fpts, grad):
6         # Code taken directly from HW1
7         for idx in range(i_begin, i_end):
8             for d in range(ndims):
9                 for j in range(nvars): # = 1
10                    sum = 0
11                    for i in range(nface):
12                        sum += fpts[i, j, idx] * op[d, i, idx]
13
14                    grad[d, j, idx] = sum
15     return self.be.make_loop(self.neles, _cal_grad)
```

4 Implementation of `_make_flux` and `_make_grad_at_face` at `inters.py`

4.1 Mathematical Implementation of `_make_flux`

In orthogonal structure, if e represents the unit vector along the direction defined by the line connecting nodes e and ef :

$$e = \frac{\mathbf{r}_{ef} - \mathbf{r}_e}{\|\mathbf{r}_{ef} - \mathbf{r}_e\|} = \frac{\mathbf{d}_{e-ef}}{d_{e-ef}} \quad (1)$$

$$(\nabla\phi \cdot e)_f = \left(\frac{\partial\phi}{\partial e} \right)_f = \frac{\phi_{ef} - \phi_e}{\|\mathbf{r}_{ef} - \mathbf{r}_e\|} = \frac{\phi_{ef} - \phi_e}{d_{e-ef}} \quad (2)$$

To linearize the flux equation in non-orthogonal structures the surface vector \mathbf{S}_f can be written as summation of two vectors \mathbf{E}_f and \mathbf{T}_f .

$$\mathbf{S}_f = \mathbf{E}_f + \mathbf{T}_f \quad (3)$$

The above equation can be rewritten as follows:

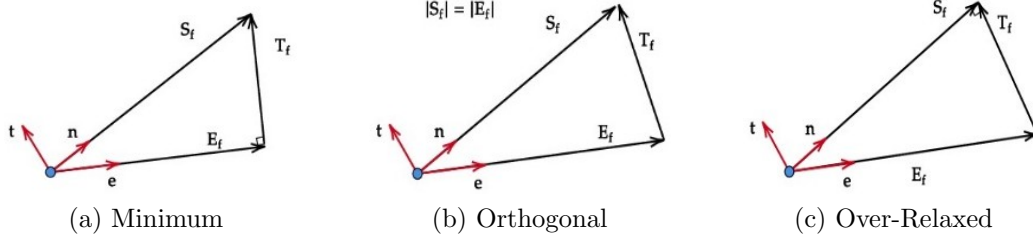
$$(\nabla\phi)_f \cdot S_f = (\nabla\phi)_f \cdot \mathbf{E}_f + (\nabla\phi)_f \cdot \mathbf{T}_f \quad (4)$$

$$(\nabla\phi)_f \cdot \mathbf{E}_f = E_f \frac{\phi_{ef} - \phi_e}{\|\mathbf{r}_{ef} - \mathbf{r}_e\|} \quad (5)$$

$$(\nabla\phi)_f \cdot S_f = E_f \frac{\phi_{ef} - \phi_e}{\|\mathbf{r}_{ef} - \mathbf{r}_e\|} + (\nabla\phi)_f \cdot \mathbf{T}_f \quad (6)$$

To calculate \mathbf{E}_f one can use the following approaches.

Approach	Implementation of \mathbf{E}_f
(a) Minimum Correction	$\mathbf{E}_f = (\mathbf{e} \cdot S_f) \mathbf{e}$
(b) Orthogonal Correction	$\mathbf{E}_f = S_f \mathbf{e}$
(c) Over-Relaxed Correction	$\mathbf{E}_f = \frac{S_f}{\mathbf{e} \cdot \mathbf{n}_f} \mathbf{e}$



Taken from lecture notes part-8 (Karakuş 2024b). With a small correction to the over-relaxed method taken from an online source (CFD-Online 2024).

4.2 Code Implementation of `_make_flux`

To implement `_make_flux` with appropriate corrections the following code is utilized for `ParabolicIntInters` class:

```

1  def _make_flux(self, nele):
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      # Inverse distance between the elements
7      inv_ef = self._rcp_dx
8      # Unit vector connecting cell centers
9      ef = self._dx_adj * inv_ef
10     # Correction method to be used
11     correction = self._correction
12     # Compiler arguments
13     array = self.be.local_array()
14     # Get compiled function of viscosity and viscous flux
15     compute_mu = self.ele0.mu_container()
16
17     def comm_flux(i_begin, i_end, muf, gradf, *uf):
18         # Parse element views (fpts, grad)
19         du = uf[:nele]
20         for idx in range(i_begin, i_end):
21             mu = compute_mu()
22             fn = 0.0
23             Sf = nf[:,idx] * sf[idx]
24             lti, lei, lfi = lt[idx], le[idx], lf[idx]
25             rti, rei, rfi = rt[idx], re[idx], rf[idx]
26
27             if correction == 'minimum':
28                 # Ef = (Sf dot e) * e
29                 Ef = dot(ef[:,idx], Sf, ndims)

```

```

30         elif correction == 'orthogonal':
31             Ef = sf[idx]
32         elif correction == 'over-relaxed':
33             Ef = (dot(Sf, Sf, ndims)/dot(Sf, ef[:, idx], ndims))
34             # https://www.cfd-online.com/Wiki/Diffusion\_term
35         else:
36             print("Wrong Correction")
37
38         Tf = Sf - Ef * ef[:, idx]
39         for k in range(nfvars):
40             fn = -1 * mu * (Ef * (du[lfi][lfi, k, lei] * inv_ef[idx])
41                             + dot(gradf[:, k, idx], Tf, ndims))
42
43             uf[lfi][lfi, k, lei] = fn
44             uf[rli][rli, k, rei] = -fn
45
46         return self.be.make_loop(self.nfpts, comm_flux)

```

Notice how there is a selector that changes E_f according to the correction method used. To save some calculations and implement the flux easier the magnitude is calculated, to obtain \mathbf{E}_f , in \mathbf{T}_f for example, the magnitude is multiplied by the \mathbf{e} vector.

The method is similar for the boundary elements however with a small addition of calling the bc function.

The code for `_make_flux` at `ParabolicBCInter` class:

```

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         # Magnitude and direction of the connecting vector
6         inv_ef = self._rcp_dx
7         ef = self._dx_adj * inv_ef
8         # Correction method to be used
9         correction = self._correction
10        # Compiler arguments
11        array = self.be.local_array()
12        # Get compiled function of viscosity and viscous flux
13        compute_mu = self.ele0.mu_container()
14        # Get bc function
15        bc = self.bc

```

```

16
17     def comm_flux(i_begin, i_end, muf, gradf, *uf):
18         # Parse element views (fpts, grad)
19         du = uf[:nele]
20         # Same as int but with no right element
21         for idx in range(i_begin, i_end):
22             Sf = nf[:, idx] * sf[idx]
23             muf = compute_mu()
24             fn = 0.0
25             lti, lei, lfi = lt[idx], le[idx], lf[idx]
26
27             if correction == 'minimum':
28                 # Ef = (Sf dot e) * e
29                 Ef = dot(Sf, ef[:, idx], ndims)
30             elif correction == 'orthogonal':
31                 Ef = sf[idx] # * ef[:, idx]
32             elif correction == 'over-relaxed':
33                 Ef = (dot(Sf, Sf, ndims)/dot(Sf, ef[:, idx], ndims))
34                 # https://www.cfd-online.com/Wiki/Diffusion_term
35             else:
36                 print("Wrong Correction")
37
38             Tf = Sf - Ef * ef[:, idx]
39             for k in range(nfvars):
40                 ur = array(nfvars)
41                 nfi = nf[:, idx]
42                 bc(du[lti][lfi, k, lei], ur, nfi)
43                 fn = -1 * muf * (Ef * (du[lti][lfi, k, lei] * inv_ef[idx])
44                             + dot(gradf[:, k, idx], Tf, ndims))
45
46             uf[lti][lfi, k, lei] = fn
47
48         return self.be.make_loop(self.nfpts, comm_flux)

```

Unlike the `ParabolicIntInters` class, this implementation additionally runs the `bc` function to obtain the values of the imaginary right element. However, whether this step is necessary or not is up to debate.

4.3 Implementation of `_make_grad_at_face`

As explained in the homework document the gradient at face is calculated using weights.

$$(\nabla q)_{face} = \omega_l (\nabla q)_{left\ element} + \omega_r (\nabla q)_{right\ element} \quad (7)$$

Which is implemented easily as the weights are already calculated previously by the `compute_weight` function.

The code is as follows for the `ParabolicIntInters` class:

```

1  def _make_grad_at_face(self, nele):
2      nvars, ndims = self.nvars, self.ndims
3      lt, le, lf = self._lidx
4      rt, re, rf = self._ridx
5      # Inverse distance between the cell center
6      weight = self._weight
7      # Stack-allocated array
8      array = self.be.local_array()
9
10     def grad_at(i_begin, i_end, gradf, *uf):
11         # Parse element views (fpts, grad)
12         du = uf[:nele]
13         gradu = uf[nele:]
14         for idx in range(i_begin, i_end):
15             lti, lei, lfi = lt[idx], le[idx], lf[idx]
16             rti, rei, rfi = rt[idx], re[idx], rf[idx]
17             for j in range(nvars):
18                 for i in range(ndims):
19                     # weight of right = 1 - weight of left
20                     gradf[i, j, idx] = weight[idx]*gradu[lti][i, j, lei]
21                     + (1 - weight[idx])*gradu[rti][i, j, rei]
22
23     return self.be.make_loop(self.nfpts, grad_at)

```

Similarly for the `ParabolicBCInters` class as no boundary gradient data is needed, the right elements gradient at cell center is set as the face gradient.

```

1  def _make_grad_at_face(self, nele):
2      nvars, ndims = self.nvars, self.ndims
3      lt, le, lf = self._lidx
4
5      # Mangitude and direction of the connecting vector
6      inv_tf = self._rcp_dx # This code was left here
7      tf = self._dx_adj * inv_tf # probably unintentionally.
8      avec = self._vec_snrm/np.einsum('ij,ij->j', tf, self._vec_snrm)
9
10     # Stack-allocated array
11     array = self.be.local_array()

```



```

12
13     def grad_at(i_begin, i_end, gradf, *uf):
14         du = uf[:nele]
15         gradu = uf[nele:]
16         # Same as int but with no right element
17         for idx in range(i_begin, i_end):
18             lti, lei, lfi = lt[idx], le[idx], lf[idx]
19             for j in range(nvars):
20                 for i in range(ndims):
21                     gradf[i, j, idx] = gradu[lti][i, j, lei]
22
23     return self.be.make_loop(self.nfpts, grad_at)

```

5 Implementation of `_make_compute_norm`

5.1 Mathematical Implementation of `_make_compute_norm`

The heat flux caused by a temperature difference $T_2 - T_1$ on a disk can be calculated analytically. The heat flux between the inner radius (r_1) and the outer radius (r_2) is determined using the following expressions.

The radial distance r_i of any element on the disk can be calculated as shown below, where x_0 and x_1 represent the x and y coordinates of the elements.

$$r_i = \sqrt{x_0^2 + x_1^2} \quad (8)$$

The radial heat flux q is calculated using the formula:

$$q = -\frac{\mu_f(T_2 - T_1)}{r_i \ln\left(\frac{r_2}{r_1}\right)} \quad (9)$$

The divergence of the temperature gradient (heat flux) enables the analysis of heat generation, dissipation or accumulation within a system. This information is essential for characterizing and optimizing heat transfer in complex geometries or materials with variable thermal conductivity.

The divergence of the heat flux, can be expressed analytically as:

$$\nabla \cdot (\mu_f \nabla \cdot \vec{q}) = -\mu_f(T_2 - T_1) \left(\frac{(x_0 - x_1)(x_0 + x_1)}{r_i \ln \left(\frac{r_2}{r_1} \right)} + \frac{x_1^2 - x_0^2}{r_i \ln \left(\frac{r_2}{r_1} \right)} \right) \quad (10)$$

With the exact solution known, the error could be calculated for radial meshes.

To measure the accuracy of the solution, L_2 norm of error will be used which is calculated as such:

$$\text{Error} = (\nabla \cdot (\mu_f \nabla \cdot \vec{q}))_{\text{numerical}} - (\nabla \cdot (\mu_f \nabla \cdot \vec{q}))_{\text{analytical}} \quad (11)$$

The total error for all elements and variables is transformed into a norm as follows:

$$L_2 \text{ norm} = \sqrt{\sum_{i=0}^n \text{Error}_i^2 \cdot V_i} \quad (12)$$

The methodology explained above is applied to the square mesh (Kershaw Mesh) with an equation for exact temperature taken from the beloved ME311 Heat Transfer course book. (Incropera 1998)

It is found that when calculated the divergence of flux is equal to 0 for square meshes after tedious implementation of the code. Or in other terms:

$$(\nabla \cdot (\mu_f \nabla \cdot \vec{q}))_{\text{analytical}} = 0 \quad (13)$$

5.2 Code Implementation of `make_compute_norm`

The code that implements the L_2 norm of error calculation for meshes with internal radius $0.1\sqrt{2}$ and outer radius $1\sqrt{2}$ with inner and outer boundary conditions of 0 and 1 is shown on the next page.

```

1  def _make_compute_norm(self):
2      # Code taken directly from HW1
3      nvars, neles, ndims = self.nvars, self.neles, self.ndims
4      vol = self._vol
5      xcs = self.xc
6      #T2g = self.cfg.get('soln-bcs-outer', 'q')
7      #T1g = self.cfg.get('soln-bcs-inner', 'q') # there was an attempt made
8      muf = 1
9
10     def run(upts):
11         err = np.zeros((nvars, neles))
12         T2, T1 = 1, 0
13         L, W = 1, 1
14         r = [0.1 * (2 ** 0.5), 1 * (2 ** 0.5)]
15         for idx in range(neles):
16             x = np.zeros(ndims)
17             for i in range(nvars):
18                 for j in range(ndims):
19                     x[j] = xcs[idx][j]
20
21                 # Regular Disk
22                 r_i = (x[0] ** 2 + x[1] ** 2) ** 0.5
23                 flux = - (muf*(T2-T1))/(r_i*np.log(r[1]/r[0]))
24                 # incropera c.5 eqn is div of flux
25                 eqn = -muf*(T2-T1)*(((x[0]-x[1])*(x[0]+x[1]))
26                                     /(np.log(r[1]/r[0])*r_i)
27                                     +((x[1]**2-x[0]**2))/(np.log(r[1]/r[0])*r_i))
28                 err[i,idx] = ((upts[i, idx] - eqn)**2)*vol[idx]
29
30                 # Kershaw
31                 '''sumx, sumy = 0, 0
32                 for f in range(1,31): # set up arbitrary infinite sum
33                     top = np.sinh(f*np.pi*x[0]/L)
34                     bot = np.sinh(f*np.pi*W/L)
35                     mid = (((-1)**(f+1))+1)*f*np.sin(f*np.pi*-x[1]/L)
36                     sumx += mid*top/bot
37                     sumy += -mid*top/bot
38                 sum = sumx + sumy
39                 eqn = ((-2*np.pi)*sum)/(L**2)
40                 err[i,idx] = ((upts[i, idx] - eqn)**2)*vol[idx]'''
41
42         norm = (np.sum(err))**0.5
43
44     return norm
45
46     return self.be.compile(run, outer=True)

```

The code also implements the Kershaw mesh exact solution, commented out for this demonstration, for meshes with square boundaries of length 1.

6 Tests and Results

In this part of the report tests of various geometries with different boundary shapes and different mesh structures will be conducted and the limits of the code will be tested to see the boundaries of the code will be explored.

To observe the iterations please visit [**here**](#).

6.1 Meshes

There are two main mesh types to be tested: Kershaw and normal meshes. Kershaw meshes have a distorted structure and they are used for to cover the needs of high local resolution in high-gradient regions. They provide more accurate results, especially when modeling shock waves in compressible flow problems. Although they use computational resources efficiently, they are more complex in terms of their creation and solution algorithms.

On the other hand, normal meshes can be structured, unstructured, or both. They are more suitable for many applications since they are relatively easy to create and the calculation processes are simpler than Kershaw counterparts, but precision may be lost in complex areas or conditions.

The very first mesh is provided by the instructor, Ali Karakuş, himself; moreover, it consists of three circles inside each other. The outer and inner circles are the boundary faces, while the middle one is there only for transfinite implementations. There are quadrilateral elements inside the middle circle and triangular elements in the outer region. It is not structured but it is symmetric in all planes on 2D axes. To obtain rapid results the provided mesh has been altered to be coarser.

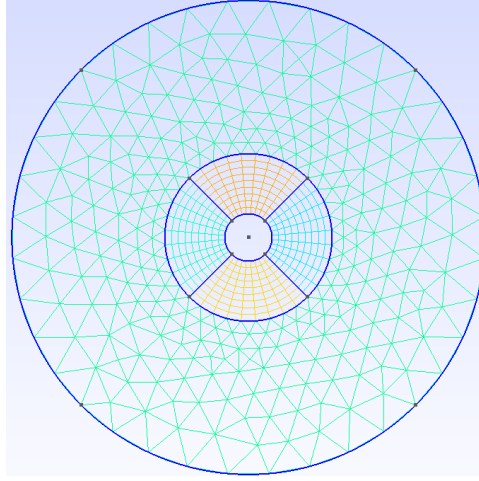


Figure 1: Provided Mesh

The second mesh consists of the same boundary geometries as the first mesh discussed above. However, unlike the first mesh, this mesh has a square in the middle. Furthermore, quadrilateral elements are pushed towards the outer region while triangular elements are pulled inside the square. This mesh is constructed to see the effect of decreasing symmetry planes on the L_2 norm.

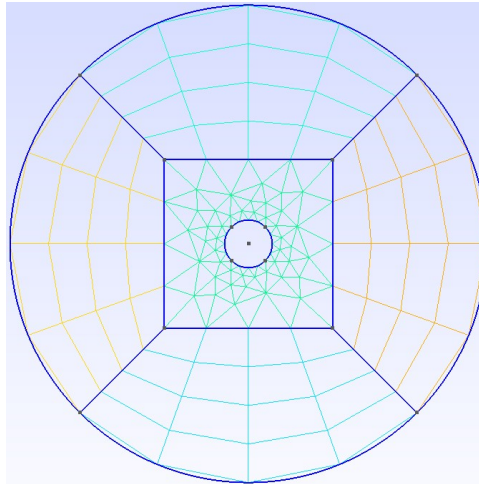


Figure 2: Circle with Square Middle

The third mesh is constructed with a quadrilateral shape in the middle. The inner and outer boundary geometries are the same as in the first mesh. The geometry resembles a compass hence it is named after it. There are triangular meshes in the inner and

quadrilateral elements outwards. This mesh is constructed to see whether symmetry is distorted between planes and affects the L_2 norm.

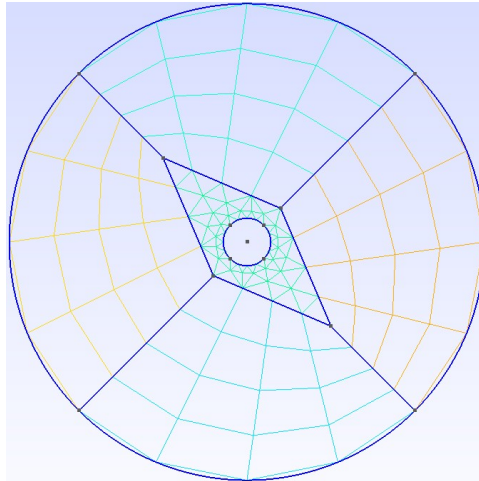


Figure 3: Compass Like Middle

The fourth mesh is the distorted version of the previous one. This mesh has properties similar to the previous one. However, it differs in terms of symmetry, as one can observe from the below figure, there are no planes of symmetry.

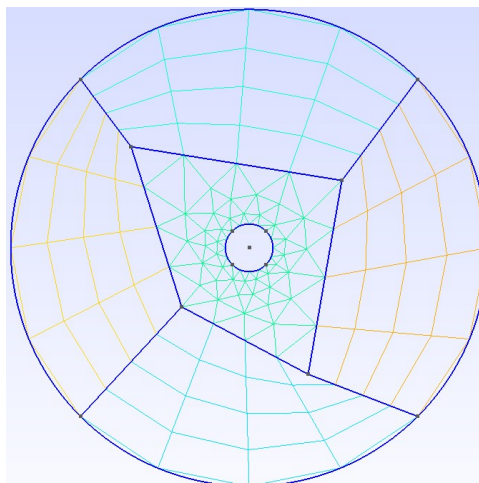


Figure 4: Distorted Compass Like Middle

From here on there will be no normal mesh types, only Kershaw mesh types will be presented. These were rather easy to implement since only α values are changing in between meshes. Respectively, values of α will be gradually increased from [0.2, 0.3, 0.5,

0.7]. It is expected to see a structured mesh when α is selected as 0.5, and our first insight is to see close L_2 norms in between α values 0.3 and 0.7. We expect to see this trend since basically they are nearly symmetric of each other.

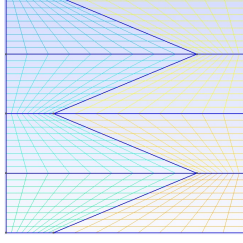


Figure 5: $\alpha = 0.2$

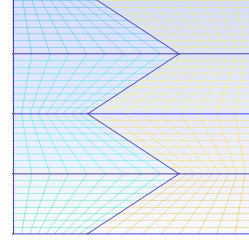


Figure 6: $\alpha = 0.3$

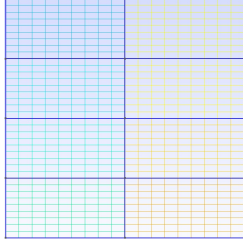


Figure 7: $\alpha = 0.5$

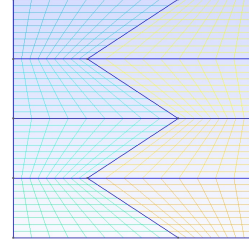


Figure 8: $\alpha = 0.7$

Furthermore, the meshes are summarized in *table 3* for easy following.

6.2 L_2 Error Norms

In this section, the L_2 error norms are tabulated with 2 different initial and boundary condition sets. One "cooling" and one "heating". This paper defines the "heating" case as the given case of $q_{ic} = 0$, $q_{inner} = 0$, $q_{outer} = 1$ and "cooling" as the opposite, meaning $q_{ic} = 1$, $q_{inner} = 1$, $q_{outer} = 0$.

Furthermore, the boundary types are set as Dirichlet as, unfortunately, the exact solution for Neumann boundary condition could not be determined without limited calculus skills and time.

The tables for cases "heating" and "cooling" are presented in the next two pages.







Mesh	L_2 Error Norm (Minimum)	L_2 Error Norm (Orthogonal)	L_2 Error Norm (Over-Relaxed)
 Figure 1	[0.000722529]	[0.000717858]	[0.000713203]
 Figure 2	[0.001150950]	[0.001132288]	[0.001113778]
 Figure 3	[0.001603565]	[0.001576355]	[0.001548625]
 Figure 4	[0.001236185]	[0.001186776]	[0.001138029]
 Figure 5	[4.73146e-11]	[4.77808e-11]	[4.84148e-11]
 Figure 6	[6.21990e-12]	[6.42256e-12]	[6.23417e-12]
 Figure 7	[3.00667e-12]	[3.00605e-12]	[3.00612e-12]
 Figure 8	[6.24074e-12]	[6.26951e-12]	[6.25994e-12]

Table 1: Results for $q_{ic} = 0$, $q_{inner} = 0$, $q_{outer} = 1$ Dirichlet



Mesh	L_2 Error Norm (Minimum)	L_2 Error Norm (Orthogonal)	L_2 Error Norm (Over-Relaxed)
 Figure 1	[0.000722529]	[0.000717858]	[0.000713203]
 Figure 2	[0.001150950]	[0.001132288]	[0.001113778]
 Figure 3	[0.001603565]	[0.001576355]	[0.001548625]
 Figure 4	[0.001236185]	[0.001186776]	[0.001138029]
 Figure 5	[4.73146e-11]	[8.81601e-12]	[8.43790e-12]
 Figure 6	[1.10656e-12]	[1.08697e-12]	1.04969e-12
 Figure 7	[4.86977e-13]	[4.86467e-13]	[4.85080e-13]
 Figure 8	[1.08052e-12]	[1.04127e-12]	[1.02194e-12]

Table 2: Results for $q_{ic} = 1$, $q_{inner} = 1$, $q_{outer} = 0$ Dirichlet

6.3 Comments

The work done follows the initial expectations. The most structured mesh, Kershaw mesh with $\alpha = 0.5$, is expected to have the smallest L_2 norm. In addition, one can go to *Table 1* or *Table 2* to see the trend of decreasing L_2 norm towards the *Kershaw mesh* with $\alpha = 0.5$.

Continuing with the tables, when a quick gaze is taken on tables, one can see a sharp increase of an order of magnitude when *test mesh 4* and *test mesh 5* are compared. This "jump" happens because the method is switched to Kershaw mesh generation. One can conclude that despite having longer computation time, Kershaw mesh generation has better results.

Another comparison can be made between normal meshes. As symmetry planes of the test mesh decreases, L_2 norm tend to increase. The smallest L_2 norm is obtained from the test mesh provided since it has infinitely large symmetry plane number.

A general comparison can be made in between *Table 1* and *Table 2*. The first table indicates a heating problem, and the second table is a cooling problem with different boundary conditions. One remark is when "unsteadiness" is concerned, it is found out that the heating case doesn't reach to steady state until the end of the time step chosen. This causes *Table 2* to have lesser L_2 norm values.

As a last comment, the comparison correction methods will be discussed. As can be seen from the *Table 1*, for the first five test meshes L_2 norm decreases as the method shifts from Minimum to Over-Relaxed. For *mesh 6 and 8*, it first increases then decreases. However for the *mesh 7* the lowest L_2 norm is reached with the Orthogonal method since the mesh is structured. When it comes to *Table 2* the trend is simple, L_2 norm decreases as the method shifts from Minimum to Over-Relaxed, and also it decreases as mesh type shifts from normal to Kershaw and unstructured Kershaw to structured Kershaw.

References

- Baygeldi-Kaya-Kakilli (2024). *Homework 1 Report*. Last accessed 27/12/2024. URL: <https://github.com/uobaygeldi/ME485-HWs/blob/main/HW1%20Report.pdf>.
- CFD-Online (2024). *Discretisation of the Diffusion Term*. Last accessed 27/12/2024. URL: https://www.cfd-online.com/Wiki/Diffusion_term.
- Incropera (1998). *Fundamentals of Heat and Mass Transfer*. Willey.
- Karakuş, Ali (2024a). *ME485 mefm HW2 Code*. Last accessed 27/12/2024. URL: <https://github.com/AliKarakus/me485-HWs/tree/HW2>.
- (2024b). *ME485 Lecture Notes*. Last accessed 27/12/2024. URL: <https://odtuclass2024f.metu.edu.tr/course/view.php?id=2835>.

A Mesh Table









Mesh	Element Count	Node Count	Reason
 Figure 1	1011	626	Default mesh
 Figure 2	261	141	Multiple axis of symmetry with both element types.
 Figure 3	221	121	Two axis of symmetry with both element types.
 Figure 4	249	135	No axis of symmetry with both element types.
 Figure 5	861	703	Kershaw mesh with $\alpha = 0.2$
 Figure 6	861	703	Kershaw mesh with $\alpha = 0.3$
 Figure 7	861	703	Kershaw mesh with $\alpha = 0.5$, structured mesh
 Figure 8	861	703	Kershaw mesh with $\alpha = 0.7$

Table 3: Statistics of Meshes