



ODTÜ METU

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

Homework 1

Uğur Ozan Baygeldi

Ali Kaya

Onur Kakilli

Group 15

1 Introduction

This report is on the first homework of the course ME485, Computational Fluid Dynamics Using Finite Volume Method. The homework requests some implementations of gradient methods to *mem*.

2 Roadmap

When the file `grad_test.py` is ran, the code eventually runs the file `system.py`. As a result, to efficiently complete the given assignment, commands were implemented according to the given order in `system.py`.

- Implement `_make_compute_fpts` at `elements.py`
- Implement `_make_delu` and `_make_avg` to `GradIntInters` and `GradBCInters` classes at `inters.py`
- Implement `_make_grad_gg` and `_make_grad_ls` with `_grad_operator` at `elements.py`
- Implement `compute_L2_norm` at `elements.py`

Writing the code this way allowed for an easier understanding of the working principles of *mem* (and prompted fewer visits to G-304 and G-157).

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

The function `_make_compute_fpts` essentially sets the center values of variables to all of the faces of an element. To achieve this 3 for loops are used iterating over all elements for all variables and all faces of the current element. Iterating over each element is handled by the backend thus in the code no there is no `self.neles` but instead `i_begin` and `i_end` is used.

The code is as follows:

```
1  def _make_compute_fpts(self):
2      nvars, nface = self.nvars, self.nface
3
4      def _compute_fpts(i_begin, i_end, upts, fpts):
5          # Copy upts to fpts
6          # fpts = np.empty((self.nface, self.nvars, self.neles))
7          # print(np.shape(upts))
8          for idx in range(i_begin, i_end):
9              for j in range(nvars):
10                 for i in range(nface):
11                     fpts[i, j, idx] = upts[j, idx]
12
13     return self.be.make_loop(self.neles, _compute_fpts)
```

For easier troubleshooting comments are used depicting how `fpts` is constructed. In addition, `np.shape()` was used extensively to figure out the structure of arrays that are not obvious by creating a test mesh with 9 elements. If the output of the shape was (1,9) for example, it is understood that the first argument is variable and the second was element id. This method was employed to figure out the arguments of `upts`. In further code blocks for this report, these tricks will be removed to de-clutter the code.

The code loops over; the elements with the parameter `idx`, variables with the parameter `j`, and faces of the current element with the parameter `i`. Then, for each loop, sets the appropriate index in `fpts` to the element cell center value.

4 Implementation of `_make_delu` and `_make_avg` to `GradIntInter`s and `GradBCInter`s at `inters.py`

The functions, `_make_delu` and `_make_avg` are crucial for implementing the Least-Squares and Weighted Least-Squares method, and the Green-Gauss-Cell-Based method respectively. Essentially these functions, when called, subtract or average the values of the neighbor faces of neighbor elements, with special care for the boundary conditions for the functions implemented to the `GradBCInter`s class.

4.1 Implementation of `_make_delu`

For the `GradIntInters` class the following code implements `_make_delu`:

```
1  def _make_delu(self):
2      nvars = self.nvars
3      lt, le, lf = self._lidx
4      rt, re, rf = self._ridx
5
6      def compute_delu(i_begin, i_end, *uf):
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 jdx in range(nvars):
12                 ul = uf[lti][lfi, jdx, lei]
13                 ur = uf[rti][rfi, jdx, rei]
14
15                 delu = ur - ul
16
17                 uf[lti][lfi, jdx, lei] = delu
18                 uf[rti][rfi, jdx, rei] = -delu
19
20     return self.be.make_loop(self.nfpts, compute_delu)
```

The code takes the information of the main element named the left element `ul`, and the adjacent element named the right element `ur`. Then for each variable, finds the difference of the faces and updates the interfaces. The left face gets updated to the value of Δu while the right face gets $-\Delta u$, where u stands for an arbitrary variable. The change in sign is caused by u being a vector.

For the boundary conditions, the procedure is similar. The main difference is that for the boundary condition, the right element is a predefined element constructed at `bcs.py`. The code at `bcs.py` simply takes the boundary conditions from the `.ini` file and applies according to specified type, creating a "pretend" right element for calculation purposes.

For the GradBCInters class the following code implements `_make_delu`:

```

1  def _make_delu(self):
2      nvars = self.nvars
3      lt, le, lf = self._lidx
4      nf = self._vec_snorm
5
6      bc = self.bc
7      array = self.be.local_array()
8
9      def compute_delu(i_begin, i_end, *uf):
10         for idx in range(i_begin, i_end):
11             ur = array(nvars)
12             nfi = nf[:, idx]
13             lti, lfi, lei = lt[idx], lf[idx], le[idx]
14
15             ul = uf[lti][lfi, :, lei]
16             bc(ul, ur, nfi)
17
18             delu = ur - ul
19
20             uf[lti][lfi, :, lei] = delu
21
22         return self.be.make_loop(self.nfpts, compute_delu)

```

Since there is no "proper" right element, setting $-\Delta u$ value to the right face is skipped. Only the left face of the intersection is updated.

4.2 Implementation of `_make_avg`

Similar to the `_make_delu` functions, `_make_avg` employs the same method. The main difference between the two is, obviously, `_make_avg` takes the averages of the faces and assigns them to each side of the face equally.

A simple change in the operation between the two faces is enough to implement `_make_avg` for the intersection and boundary condition classes by swapping the difference operation to an averaging operation and setting the same value on both sides.

For the GradIntInters class the following code implements `_make_avgu`:

```

1  def _make_avgu(self):
2      nvars = self.nvars
3      lt, le, lf = self._lidx
4      rt, re, rf = self._ridx
5
6      def compute_avgu(i_begin, i_end, *uf):
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 jdx in range(nvars):
12                 ul = uf[lti][lfi, jdx, lei]
13                 ur = uf[rti][rfi, jdx, rei]
14
15                 avgu = (ul+ur)/2
16
17                 uf[lti][lfi, jdx, lei] = avgu
18                 uf[rti][rfi, jdx, rei] = avgu
19
20     return self.be.make_loop(self.nfpts, compute_avgu)

```

Similarly for the GradBCInters class the following code implements `_make_avgu`:

```

1  def _make_avgu(self):
2      nvars = self.nvars
3      lt, le, lf = self._lidx
4      nf = self._vec_snorm
5      bc = self.bc
6      array = self.be.local_array()
7
8      def compute_avgu(i_begin, i_end, *uf):
9          for idx in range(i_begin, i_end):
10              ur = array(nvars)
11              nfi = nf[:, idx]
12              lti, lfi, lei = lt[idx], lf[idx], le[idx]
13              ul = uf[lti][lfi, :, lei]
14              bc(ul, ur, nfi)
15
16              avgu = (ur + ul)/2
17
18              uf[lti][lfi, :, lei] = avgu
19
20     return self.be.make_loop(self.nfpts, compute_avgu)

```

4.3 Possible Problems of the Method

The method assumes that the element has one boundary face. For rectangular grids with quad meshes, it is possible to encounter elements with more than one boundary face, for cases like those, a better implementation could be prepared.

5 Implementation of `_make_grad_gg` and `_make_grad_ls` with `_grad_operator` at `elements.py`

5.1 The Green-Gauss Cell Based Method

The Green-Gauss method is a numerical tool used in computational fluid dynamics (CFD) to compute the gradient. This gradient can be of the type of properties such as the temperature, pressure, or velocity components at the control volume. It can be implemented on centers or nodes based on the method used to obtain the gradient. Green-Gauss is based on the divergence theorem which converts a vector field through a closed surface to a divergence of the field within the volume encapsulated by that surface. In the Green-Gauss approach, the gradient of a scalar field at the centroid of a control volume is approximated by integrating over the control volume's faces. The method assumes that the variation of the scalar field is linear within the control volume, and this is the reason why it is fast and straightforward to implement. In practice, the Green-Gauss method computes gradients by summing contributions from all the faces of a control volume. For each face, the scalar field values at the neighboring cell centers are used to calculate the value of the field at the face centroid, often interpolated linearly.

Normally the flux through each face is then weighted by the face's area and its normal vector, however, the scope of this homework is not to make the weighted Green Gauss method. Then the resulting gradients must be divided by volume to maintain conservation properties and accuracy.

5.2 Mathematical Implementation of the Green-Gauss Cell Based Method

By utilizing the Divergence Theorem, one can convert the volume integral to surface integral as shown below.

$$\int_V (\nabla \cdot \phi) dV = \oint_S (\phi \cdot n) dS \quad (1)$$

Then one may utilize the above equality to find the gradient as given below, where $\nabla\phi_e$ is gradient at the center of the element, V_e is the volume of the respective element, ϕ_f is the face values, n_f is the unit normal of the face, N_f is the number of faces and S_f is the surface area of the respective element's respective face.

$$\nabla\phi_e = \frac{1}{V_e} \sum_{i=1}^{N_f} (\phi_f n_f) S_f \quad (2)$$

5.3 Implementation of `_make_grad_gg`

When `system.py` is observed, it can be seen that before calling `_make_grad_gg`, `mesh.m` calls `compute_avgu` for boundary and element interfaces. From *section 4* of this report, recall that `compute_avgu` which calls `_make_avgu` adjusts the flux point array, `fpts`, such that every side of every element has the value of neighbor faces values averaged.

To calculate the gradient, the code loops over every element, for every element another loop, loops over every dimension, for every dimension another loop, loops over every variable, and lastly for every variable another loop loops over every face of the element. For every variable, a sum is calculated by adding up all the values of faces. The sum is then simply placed at the appropriate spot in the `grad` array. To be used later on while calculating the error or when necessary for further assignments.

The following code implements `_make_grad_gg`

```
1  def _make_grad_gg(self):
2
3      nface, ndims, nvars = self.nface, self.ndims, self.nvars
4      # Normal vector and volume
5      snorm_mag = self._mag_snorm_fpts # face / elemid
6      snorm_vec = np.rollaxis(self._vec_snorm_fpts, 2) # x,y / face / elemid
7      vol      = self._vol # elemid
8      def _cal_grad(i_begin, i_end, fpts, grad):
9          # Elementwise loop starts here
10         for idx in range(i_begin, i_end):
11             for d in range(ndims):
12                 for j in range(nvars): # = 1
13                     sum = 0
14                     for i in range(nface):
15                         sum += fpts[i,j,idx]*snorm_mag[i,idx]*snorm_vec[d,i,idx]
16
17                     grad[d,j,idx] = (1/vol[idx])*sum
18
19     return self.be.make_loop(self.neles, _cal_grad)
```

5.4 Comments for the Green-Gauss Cell Based Method

While the Green-Gauss method is widely used due to its simplicity, its accuracy can be affected in cases where the mesh is highly distorted which will be inspected in detail with different mesh resolutions and types. Furthermore, if the scalar field varies significantly, the accuracy of this method drops severely. Furthermore, the limitations previously mentioned on *section 4.3* also applies as the method calls `_make_avgu` at the boundaries.

5.5 The Least Squares Gradient Method

The least squares method is another technique used in CFD solvers for calculating gradients. Unlike the Green-Gauss method, which was explained previously, the least squares method directly minimizes the error between the computed gradients and the differences in field values between neighboring cells. This approach works better than Green Gauss in both structured and unstructured meshes. Thus, making it more suitable for complex geometries that require complex mesh structures. The method creates

a system of equations using the field values at neighboring nodes or cell centers, and it constructs an over-determined system with more linearly independent equations than unknown quantities.

In practice, the least squares gradient evaluation starts by selecting a set of neighboring cells for a given control volume. For each neighbor, the solver calculates the difference in the scalar or vector field value and divides it by the vector distance between the cell centers. These differences are used to construct a system of equations representing the gradient components.

5.6 Mathematical Implementation of the Least Squares Method

One can calculate the gradient of elements at the center by using the following equation:

$$\phi_{e_i} = \phi_{e_0} + (\nabla\phi)_{e_0}(r_{ei} - r_{e0}) \quad (3)$$

But when using this approach, there will be three different gradient values for the same center point. Since this is physically impossible, one needs to choose the best available one. To do so the following equation where n is the number of elements may be utilized.

$$A = \begin{bmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \\ \vdots & \vdots \\ x_n - x_0 & y_n - y_0 \end{bmatrix} \quad x = \left[\frac{\partial\phi}{\partial x} \right]_{e_0} \quad b = \begin{bmatrix} \phi_{e_1} - \phi_{e_0} \\ \phi_{e_2} - \phi_{e_0} \\ \vdots \\ \phi_{e_n} - \phi_{e_0} \end{bmatrix} \quad (4)$$

$$Ax = b \quad (5)$$

Since matrix A is not square matrix, multiplying both sides with A^T is not an option. Instead, one may use pseudo inverse method.

$$x = (A^T A)^{-1} A^T b \quad (6)$$

5.7 Implementation of `_make_grad_ls` with `_grad_operator`

To begin with the implementation of the Least Squares Gradient method, as the code requires the operator, `op`, as an input, one must first implement `_grad_operator`. Thankfully, the `_grad_operator` code could be sourced from the `base\elements.py` file, as an added bonus, this operator also includes weight options and a hybrid method. Essentially `_grad_operator` creates the A matrix and then takes the inverse of it using the least squares method.

The code for `_grad_operator` (with proper citation) is as follows:

```
1  def _grad_operator(self):
2      # Difference of displacement vector (cell to cell)
3      # (Nfaces, Nelements, dim) -> (dim, Nfaces, Nelements)
4      dxc = np.rollaxis(self.dxc, 2)
5      # (Nfaces, Nelements)
6      distance = np.linalg.norm(dxc, axis=0)
7
8      # Code taken directly from base\elements.py, thanks hocam :)
9
10     # Normal vector and volume
11     snorm_mag = self._mag_snorm_fpts
12     snorm_vec = np.rollaxis(self._vec_snorm_fpts, 2)
13     vol = self._vol
14
15     if self._grad_method == 'least-square':
16         beta, w = 1.0, 1.0
17     elif self._grad_method == 'weighted-least-square':
18         # Inverse distance weight
19         beta, w = 1.0, 1 / distance ** 2
20     elif self._grad_method == 'green-gauss':
21         beta, w = 0.0, 1.0
22     elif self._grad_method == 'hybrid':
23         # Shima et al., Green{Gauss/Weighted-Least-Squares
24         # Hybrid Gradient Reconstruction for
25         # Arbitrary Polyhedra Unstructured Grids, AIAA J., 2013
26         # WLSQ(G)
27         dxf = self.dxf.swapaxes(0, 1)
28
29         dxcn = np.einsum('ijk,ijk->jk', dxc, snorm_vec)
30         dxfn = np.einsum('ijk,ijk->jk', dxf, snorm_vec)
31
```

```

32         w = (2 * dxfn / dxcn) ** 2 * snorm_mag / distance
33
34         # Compute blending function (GLSQ)
35         ar = 2 * np.linalg.norm(self.dxf, axis=1).max(
36             axis=0)*snorm_mag.max(axis=0) / vol
37         beta = np.minimum(1, 2 / ar)
38     else:
39         raise ValueError("Invalid gradient method : ", self._grad_method)
40
41     # Scaled dxc vector
42     dxcs = dxc * np.sqrt(w)
43
44     # Least square matrix [dx*dy] and its inverse
45     lsq = np.array([[np.einsum('ij,ij->j', x, y)
46                       for y in dxcs] for x in dxcs])
47
48     # Hybrid type of Ax=b
49     A = beta * lsq + 2 * (1 - beta) * vol * np.eye(self.ndims)[: , : , None]
50     b = beta * dxc * w + 2 * (1 - beta) * 0.5 * snorm_vec * snorm_mag
51
52     # Solve Ax=b
53     op = np.linalg.solve(np.rollaxis(
54         A, axis=2), np.rollaxis(b, axis=2)).transpose(1, 2, 0)
55
56     return op

```

The grad operator, as explained previously in *section 5.6* is simply the inverse of the matrix A in the equation $Ab = x$. With $(A^T A)^{-1} A^T$ known, it is easy to compute the gradient if b is known. Tracing the steps in `system.py` it is found that `_make_delu` is called which recalling from *section 4* updates `fpts` with the difference of faces, essentially converting `fpts` to the matrix b . Then, similar to the Green-Gauss Cell Based Method, the gradient is calculated by means of nested for loops iterating over the elements, dimensions, variables, and faces. Multiplying the appropriate element of the `op`, which is the grad operator, with the appropriate element of `fpts`. Summing the results for every face of the element, as the values obtained are vectoral with respect to the dimension, then placing the sum to the appropriate location in `grad` to obtain the gradient.

The following code implements `_make_grad_ls`:

```
1  def _make_grad_ls(self):
2      nface, ndims, nvars = self.nface, self.ndims, self.nvars
3      # Gradient operator
4      op = self._grad_operator
5      def _cal_grad(i_begin, i_end, fpts, grad):
6          # Elementwiseloop
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
16      # Compile the function
17      return self.be.make_loop(self.neles, _cal_grad)
```

5.8 Comments for the Least Squares Gradient Method

Solvers typically apply a weighting scheme based on the distance or cell connectivity to account for mesh irregularities, on the other hand, our solver didn't do this for now. This will create accuracy problems while calculating gradient which will be discussed with different examples later.

Once the system is solved, the resulting gradient at the cell center is still more accurate than Green-Gauss in distorted or skewed meshes. However, this method is computationally more included due to the more steps of linear operators like matrix inversion or factorization. Still, its robustness and improved accuracy for non-uniform meshes make it a preferred choice in modern CFD.

6 Implementation of `compute_L2_norm` at `elements.py`

The L_2 error norm is used widely in the world of CFD to determine how accurate the solver is.

6.1 Mathematical Implementation of the L_2 error norm

Assume X is an $N \times 2$ matrix, such that all x and y coordinates of all elements are stored in it. Where N is the number of elements.

$$X = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \quad (7)$$

To calculate the L_2 error norm, first one needs to calculate the exact gradient. To do so one may utilize the following equations for the given test gradient. Where subscript i shows i^{th} element.

$$\nabla_{exact_i} = \begin{bmatrix} \frac{x_i}{\sqrt{x_i^2 + y_i^2}} & \frac{y_i}{\sqrt{x_i^2 + y_i^2}} \end{bmatrix} \quad (8)$$

After calculating ∇_{exact} at the cell center, the error can be calculated by expanding the result into the whole cell volume by multiplying with the element volume. Now one can calculate the L_2 error norm.

$$L_2 = \sqrt{\left[\sum_{i=1}^n (\nabla_{computed_i} - \nabla_{exact_i}) V_i \right]^2} \quad (9)$$

6.2 Implementation of `compute_L2_norm`

To implement the L_2 error norm as mentioned in the previous subsection, one must first obtain the exact values. These exact values require the exact gradient which sadly cannot be calculated numerically and unless a symbolic differentiator is used has to be hard coded into the code. To make the implementation seamless and allow for higher dimensions, an array is used. The code once again loops through every element for every variable and dimension recording the cell center values. Then with the hard coded gradient creates the exact gradient matrix. After exact gradient is obtained, the error matrix is then formed by subtracting the exact gradient from the found gradient and multiplying

the result with the element volume. Lastly to obtain the L_2 error norm, the L_2 norm of the error matrix is obtained through `np.linalg.norm(err, axis=2)`.

The code for `compute_L2_norm` is as follows:

```

1  def compute_L2_norm(self):
2      # adjust faces to eles
3      nvars, neles, ndims = self.nvars, self.neles, self.ndims
4      vol = self._vol
5      xcs = self.xc # Cell center point (x,y)
6      exactGrad = np.zeros((self.ndims, self.nvars, self.neles))
7      err = np.zeros((self.ndims, self.nvars, self.neles))
8      for idx in range(neles):
9          x = np.zeros(ndims)
10         for i in range(nvars):
11             for j in range(ndims):
12                 x[j] = xcs[idx][j]
13                 eqn = [x[0]/math.sqrt(x[0]*x[0]+x[1]*x[1]), x[1]/math.sqrt(
14                     x[0]*x[0] + x[1]*x[1])] # sqrt(x^2 + y^2)
15                 #eqn = [2*x[0], 2*x[1]] # x^2 + y^2
16                 #eqn = [32*(x[0]**31), 32*(x[1]**31)] # x^32 + y^32
17                 #eqn = [-math.sin(x[0]**2+x[1]**2)*2*x[0],
18                     # -math.sin(x[0]**2+x[1]**2)*2*x[1]]
19                 for j in range(ndims):
20                     exactGrad[j, i, idx] = eqn[j]
21                     err[j, i, idx] = (self.grad[j, i, idx] - exactGrad[j, i, idx]
22                         )*vol[idx]
23
24         resid = np.linalg.norm(err, axis=2)
25     return resid

```

7 Tests and Results

7.1 Input Meshes

The first test mesh is the mesh provided with the assignment, it is a mixed mesh with triangle and quadrilateral meshes. This mesh is easy to understand and provides a basic understanding of the `gmsh` structure. To ease the test-case generation, further meshes were created with the same boundary locations as the `q` value changes at the boundaries. Also considering that the report covers different equations for `q` setting the boundary

locations the same eliminates unnecessary calculations.

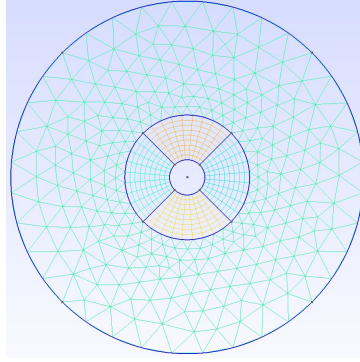


Figure 1: Provided Mesh (grad.msh)

Next mesh used is similar to the first one however uses only quadrilateral meshes. This example uses 100 uniformly placed nodes along the edges. The Python file that generates this mesh has an option to change the amount of uniformly placed nodes and this mesh will be used later on for comments as one would expect the L_2 error norm to get smaller with finer mesh.

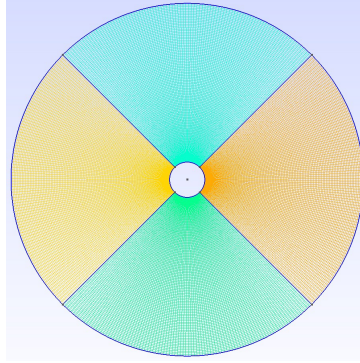


Figure 2: Finer Quad Mesh (grads.msh)

Continuing with different mesh structures, the next mesh uses mixed mesh element generation method. This particular mesh is constructed to check whether the implemented gradient calculation methods work on nonconventional meshes using geometry. It consists of a circle for the inner boundary, two triangle-like regions, eight triangles, two trapezoidal, and four regions between the outer circle and the concave octagon inside.

Moreover, the first two triangle-like geometries and the last region mentioned are constructed by triangle elements with no transfinite function implemented. Other regions consist of quadrilateral mesh types.

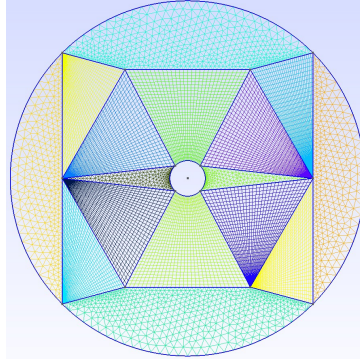


Figure 3: Weird Mesh 1 (gradok1.msh)

This particular mesh consists of highly structured rectangular mesh elements. It consists of eight subsections to create these tightly packed elements. North, east south, and west sections are divided to 9x10 and others are 10x10. This mesh is constructed to see whether structured meshes will result in lower L_2 norms for different cases.

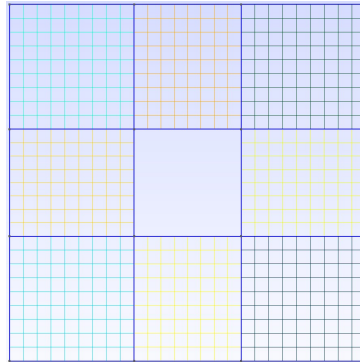


Figure 4: Box in Box Structured (gradok2.msh)

The very next mesh is similar to the previous one, it consists of two squares one inside the other and with mixed elements. Yet, this time rather than being structured, it is unstructured, and in two ways, north and south, it consists of triangular elements while the other two are quadrilateral.

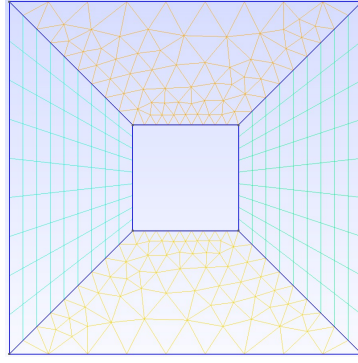


Figure 5: Box in Box Unstructured (gradok3.msh)

This mesh and the one after it were constructed with unusual geometries. The reason why is to see the effect of distorted geometries on the L_2 norm. Also, the orthogonality of elements is another source of error while calculating the gradient. One can see both meshes below. The first one which is called "Pointy Star in Box" consists of a concave octagon for the inner boundary, a circle to construct transfinite curves, and a square for the outer boundary.

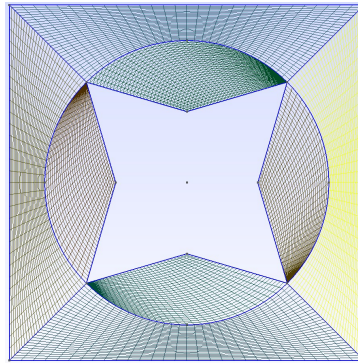


Figure 6: Pointy Star in Box (gradak1.msh)

The other mesh is called "The Eye in Box" and consists of four circle arcs with different radii and center points, a full circle centered at the origin, and lastly the outer square for the outer boundary.

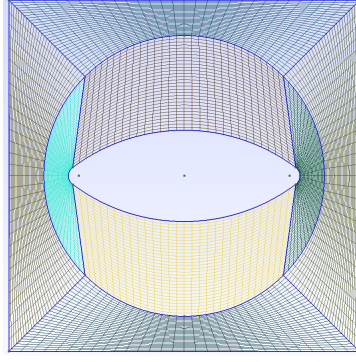


Figure 7: The Eye in Box (gradak2.msh)

7.2 Input Equations

- $q = \sqrt{x^2 + y^2}$: This equation is selected since it was given in the homework code itself. It is used to calculate the L_2 norm in all of the test meshes to get comparative results.
- $q = x^2 + y^2$: This is the previous equation without square root operator. This is selected since it was well defined in all boundaries.
- $q = \cos(x^2 + y^2)$: This equation is selected to see whether the code implemented works for trigonometric functions or not. *Cos* is selected on purpose since it is symmetric concerning the y-axis.
- $q = x^{32} + y^{32}$: This high-order function is selected for testing the limits of the code. Since it results in high numbers both numeric errors and method errors are enlarged.

7.3 L_2 Error Norms

Note that the results calculate the L_2 error separately for each type of element. As a result, there are more than one result for meshes with more than one type of element. Furthermore, the results are reported in the format of $[[x],[y]]$ with the first reported being triangular elements (if available) while the second line reports the quadrilateral elements (again, if available).

7.4 Results with Neumann Type Boundary Conditions

Neumann boundary condition enforces no flux at boundaries which means that in the boundaries there is a "pretend" element with the same value of the left element. With no flux at the boundaries, Neumann-type boundary conditions allow for easy test case setup.

Furthermore, the Neumann boundary condition eliminates the errors obtained during the boundary flux calculation phase of the test-case generation.

The results are tabulated below and on the next pages in tables 1 to 4:








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[0.00240548],[0.00241336]] [[0.07336319],[0.07294885]]	[[0.00246433],[0.00246418]] [[0.06826314],[0.06729497]]	[[0.00239596],[0.00239596]] [[0.06806001],[0.06765304]]
 Figure 2	[[0.0020397],[0.0020397]]	[[0.00203185],[0.00203185]]	[[0.0020316],[0.0020316]]
 Figure 3	[[0.00210492],[0.00149657]] [[0.01236622],[0.01227495]]	[[0.00082997],[0.00054637]] [[0.01092133],[0.01090415]]	[[0.00036063],[0.00068709]] [[0.01088902],[0.01081846]]
 Figure 4	[[0.0054402],[0.0054402]]	[[0.00543991],[0.00543991]]	[[0.00543995],[0.00543995]]
 Figure 5	[[0.01048043],[0.00367857]] [[0.00505206],[0.01158873]]	[[0.00818032],[0.00074892]] [[0.0011861],[0.01094454]]	[[0.00926219],[0.00122]] [[0.0009269],[0.0109231]]
 Figure 6	[[0.10761094],[0.10719521]] [[0.01001063],[0.01608607]]	[[0.06580105],[0.05884946]] [[0.00266881],[0.01688081]]	[[0.08054357],[0.0738357]] [[0.00425521],[0.01401713]]
 Figure 7	[[0.10955121],[0.11720963]]	[[0.05365295],[0.07589494]]	[[0.07224261],[0.08830517]]

Table 1: Results for $q = \sqrt{x^2 + y^2}$ Neumann








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[0.00140965],[0.00142458]] [[0.19672191],[0.19469873]]	[[0.00083029],[0.00082941]] [[0.18514975],[0.1825498]]	[[0.00069086],[0.0006908]] [[0.18469784],[0.18361626]]
 Figure 2	[[0.00574039],[0.00574039]]	[[0.00571564],[0.00571564]]	[[0.00571492],[0.00571492]]
 Figure 3	[[0.00390851],[0.00229017]] [[0.03404243],[0.03366623]]	[[0.00028672],[0.00019171]] [[0.0305426],[0.03048732]]	[[0.00011725],[0.00020285]] [[0.03045901],[0.03025583]]
 Figure 4	[[0.00546429],[0.00546429]]	[[0.0054642],[0.0054642]]	[[0.00546418],[0.00546418]]
 Figure 5	[[0.01146731],[0.00422454]] [[0.00523682],[0.01167767]]	[[0.00873108],[0.00089568]] [[0.00096221],[0.0110435]]	[[0.0100125],[0.00128757]] [[0.00078837],[0.01112919]]
 Figure 6	[[1.31401775],[1.31253684]] [[0.07762136],[0.11023263]]	[[0.64263714],[0.61776353]] [[0.01468941],[0.10029174]]	[[0.83643376],[0.8160706]] [[0.02239149],[0.0772916]]
 Figure 7	[[1.31916359],[1.32371419]]	[[0.59665498],[0.62363335]]	[[0.80742175],[0.8235998]]

Table 2: Results for $q = x^2 + y^2$ Neumann








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[0.00037888],[0.00038329]] [[0.19058115],[0.18828494]]	[[6.5388e-05],[6.5793e-05]] [[0.17906887],[0.17648487]]	[[3.5068e-05],[3.5340e-05]] [[0.17853048],[0.17742921]]
 Figure 2	[[0.00522279],[0.00522279]]	[[0.00519991],[0.00519991]]	[[0.00519924],[0.00519924]]
 Figure 3	[[0.00323153],[0.00179723]] [[0.03178523],[0.03142236]]	[[0.00011116],[0.00012576]] [[0.02831751],[0.02827726]]	[[7.5896e-05],[6.4593e-05]] [[0.02823049],[0.02805004]]
 Figure 4	[[0.00179106],[0.00179106]]	[[0.00179106],[0.00179106]]	[[0.00179105],[0.00179105]]
 Figure 5	[[0.0038207],[0.00158006]] [[0.00176743],[0.00313644]]	[[0.00290303],[0.00052983]] [[0.00033844],[0.00293117]]	[[0.00334377],[0.00047014]] [[0.00027598],[0.00304322]]
 Figure 6	[[1.42608863],[1.42529982]] [[0.06092356],[0.09625521]]	[[1.4520296],[1.44568491]] [[0.04909822],[0.10070024]]	[[1.27655278],[1.26899789]] [[0.05077559],[0.12243265]]
 Figure 7	[[1.43300951],[1.43431454]]	[[1.4486257],[1.45477046]]	[[1.27167525],[1.27577778]]

Table 3: Results for $q = \cos(x^2 + y^2)$ Neumann








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[5.8831e-11],[5.9212e-11]] [[11617.5792],[11598.3844]]	[[5.2688e-11],[5.3015e-11]] [[11551.6226],[11511.8233]]	[[4.6176e-11],[4.6361e-11]] [[11626.8597],[11741.9614]]
 Figure 2	[[1313.4353],[1313.4353]]	[[1308.8182],[1308.8182]]	[[1308.8435],[1308.8435]]
 Figure 3	[[0.04038635],[0.00961823]] [[5459.4218],[5675.2439]]	[[0.01646438],[0.00522647]] [[5403.7340],[5582.2484]]	[[0.01743961],[0.00520740]] [[5424.7127],[5533.0589]]
 Figure 4	[[3.8944e-11],[3.8944e-11]]	[[3.8944e-11],[3.8944e-11]]	[[3.8944e-11],[3.8944e-11]]
 Figure 5	[[6.4849e-11],[8.8478e-13]] [[6.0220e-12],[4.2501e-11]]	[[6.5355e-11],[1.9387e-12]] [[7.5760e-12],[4.3563e-11]]	[[6.6582e-11],[4.8111e-12]] [[7.8197e-12],[4.5038e-11]]
 Figure 6	[[9.3061e+21],[9.3061e+21]] [[1.0466e+17],[2.8341e+17]]	[[9.7141e+21],[9.7141e+21]] [[1.0106e+17],[2.2595e+17]]	[[1.1158e+22],[1.1158e+22]] [[9.5219e+16],[3.8537e+17]]
 Figure 7	[[9.3061e+21],[9.3061e+21]]	[[9.7141e+21],[9.7141e+21]]	[[1.1158e+22],[1.1158e+22]]

Table 4: Results for $q = x^{32} + y^{32}$ Neumann

7.5 Results with Dirichlet Type Boundary Conditions

These results were obtained by using the Dirichlet condition. For the inner and outer fluxes the initial condition equation was evaluated at the location of the boundaries for uncommon shaped boundaries the equation was evaluated at different locations on the boundary and simply averaged to get a q value.

This introduces a lot of error for uncommon boundary shapes and/or unsymmetrical variable equations. The error introduced is obvious when one compares the Dirichlet boundary condition L_2 error norms of the same mesh with the Neumann boundary condition L_2 error norms.

The results are tabulated on the next pages in tables 5 to 8:








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[0.00161697],[0.00162867]] [[0.04102047],[0.04169986]]	[[0.00139097],[0.00139071]] [[0.03299403],[0.03254984]]	[[0.00126205],[0.00126205]] [[0.03287267],[0.0326414]]
 Figure 2	[[0.00101287],[0.00101287]]	[[0.00100936],[0.00100936]]	[[0.00100898],[0.00100898]]
 Figure 3	[[0.00209438],[0.00137893]] [[0.00784781],[0.00788279]]	[[0.00042374],[0.00028083]] [[0.00539358],[0.00538197]]	[[0.00018216],[0.00034561]] [[0.00537691],[0.00533905]]
 Figure 4	[[0.0186705],[0.0186705]]	[[0.01867041],[0.01867041]]	[[0.01867042],[0.01867042]]
 Figure 5	[[0.03082349],[0.00367857]] [[0.00505206],[0.03057457]]	[[0.03074471],[0.00116142]] [[0.00290107],[0.02801405]]	[[0.03321566],[0.00459645]] [[0.0012708],[0.02808338]]
 Figure 6	[[1.90565731],[1.89260198]] [[0.09239694],[0.23073964]]	[[1.9165211],[1.89732045]] [[0.07026629],[0.23439571]]	[[2.25920684],[2.2357312]] [[0.09219926],[0.24661735]]
 Figure 7	[[1.88256696],[1.99961164]]	[[1.88497715],[1.99985048]]	[[2.22522845],[2.32094407]]

Table 5: Results for $q = \sqrt{x^2 + y^2}$ Dirichlet








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[0.00131815],[0.0013341]] [[0.105498],[0.10578135]]	[[0.0005871],[0.00058587]] [[0.08760545],[0.08643159]]	[0.00036226],[0.00036214] [[0.08733679],[0.08672503]]
 Figure 2	[[0.00285672],[0.00285672]]	[[0.00284562],[0.00284562]]	[[0.00284453],[0.00284453]]
 Figure 3	[[0.0039081],[0.00228413]] [[0.02103257],[0.02094605]]	[[0.00018318],[0.00014021]] [[0.01501804],[0.0149766]]	[[7.3987e-05],[1.1096e-04]] [[0.01497567],[0.01486289]]
 Figure 4	[[0.0246458],[0.0246458]]	[[0.0246458],[0.0246458]]	[[0.02464564],[0.02464564]]
 Figure 5	[[0.0382913],[0.00422454]] [[0.00523682],[0.03859335]]	[[0.03802043],[0.00118542]] [[0.0038781],[0.03555952]]	[[0.041346],[0.00584687]] [[0.0016699],[0.03567757]]
 Figure 6	[[23.3627812],[23.3219324]] [[0.56099016],[1.43135202]]	[[23.397633],[23.33718899]] [[0.42738759],[1.45156007]]	[[27.7407169],[27.6658563]] [[0.56895783],[1.51328272]]
 Figure 7	[[23.2838976],[23.5061674]]	[[23.2898715],[23.5079755]]	[[27.6163662],[27.7956617]]

Table 6: Results for $q = x^2 + y^2$ Dirichlet








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (wLSQ)
 Figure 1	[[90.9382921],[90.9382921]] [[0.31523253],[0.3133322]]	[[103.356936],[103.356936]] [[0.30020971],[0.29723711]]	[[102.965992],[102.965992]] [[0.3014124],[0.30211584]]
 Figure 2	[[27.4535631],[27.4535631]]	[[28.7013832],[28.7013832]]	[[28.6996371],[28.6996371]]
 Figure 3	[[21.2471590],[45.1135171]] [[45.1137279],[21.2476050]]	[[54.9763448],[38.0319801]] [[45.0246162],[21.4202675]]	[[23.9674997],[47.4004365]] [[43.8454258],[23.7877275]]
 Figure 4	[[0.00889283],[0.00889283]]	[[0.00889283],[0.00889283]]	[[0.00889283],[0.00889283]]
 Figure 5	[[0.01496869],[0.00158006]] [[0.00176743],[0.01533238]]	[[0.01475973],[0.00053461]] [[0.00163041],[0.01417951]]	[[0.01621377],[0.00236271]] [[0.0007965],[0.01425679]]
 Figure 6	[[1.99221924],[1.95506913]] [[0.13669286],[0.40386797]]	[[1.98197133],[1.9184128]] [[0.10663218],[0.42126239]]	[[2.22111504],[2.15260355]] [[0.13141051],[0.42764634]]
 Figure 7	[[1.93051196],[1.98492893]]	[[1.87797907],[1.9347084]]	[[2.13312513],[2.16706029]]

Table 7: Results for $q = \cos(x^2 + y^2)$ Dirichlet








Mesh	L_2 Error Norm (GG)	L_2 Error Norm (LSQ)	L_2 Error Norm (w-LSQ)
 Figure 1	[[5.8831e-11],[5.9212e-11]] [[14614.5980],[14761.8929]]	[[5.2688e-11],[5.3015e-11]] [[14341.5070],[14376.4434]]	[[4.6176e-11],[4.6361e-11]] [[14476.3027],[14800.5200]]
 Figure 2	[[5295.0064],[5295.0064]]	[[5271.5776],[5271.5776]]	[[5272.1033],[5272.1033]]
 Figure 3	[[0.04038635],[0.00961823]] [[10596.9569],[10652.1428]]	[[0.01646438],[0.00522647]] [[10464.7119],[10495.0431]]	[[0.01743961],[0.00520740]] [[10525.0680],[10410.7479]]
 Figure 4	[[1.7328e-11],[1.7328e-11]]	[[1.7328e-11],[1.7328e-11]]	[[1.7328e-11],[1.7328e-11]]
 Figure 5	[[3.1308e-11],[8.8478e-13]] [[6.0220e-12],[5.4728e-11]]	[[2.6098e-11],[3.3802e-12]] [[1.5130e-11],[4.9763e-11]]	[[3.7092e-11],[1.2308e-11]] [[7.1365e-12],[4.7924e-11]]
 Figure 6	[[2.5122e+22],[2.5122e+22]] [[1.0466e+17],[2.8341e+17]]	[[2.4784e+22],[2.4784e+22]] [[1.0106e+17],[2.2595e+17]]	[[3.1722e+22],[3.1722e+22]] [[9.5219e+16],[3.8537e+17]]
 Figure 7	[[2.5122e+22],[2.5122e+22]]	[[2.4784e+22],[2.4784e+22]]	[[3.1722e+22],[3.1722e+22]]

Table 8: Results for $q = x^{32} + y^{32}$ Dirichlet

7.6 Comments

In this section of the report various comments about the work done will be shared. Overall the results more or less follow the expected results. For instance, the most structural mesh, given in *figure 4*, has the smallest error norm out of all the meshes and test functions constructed by students. Furthermore; the highest error norms are achieved for all functions tested, which are the mesh structures shared at *figure 6* and *figure 7*.

Also as expected the L_2 error norm decreases as the number of elements increases. When the mesh depicted at *figure 2* is adjusted for 1000 uniformly placed nodes, the L_2 error norm drops quite noticeably for both boundary conditions.

Comparison of Dirichlet and Neumann boundary conditions is next the comment will be shared. The result of this comparison is made by comparing the last two cases of Neumann to their Dirichlet counterparts. In short, this difference is caused by the errors introduced in the step of computing the boundary value for Dirichlet. For those who are interested, the reason behind this error has already been explained in *Section 7.5*.

The next comment is the comparison of trigonometric function cases for different element types. For the boundary type Neumann, L_2 norm and element type connection is highly dependent on geometry. Inspecting the shared results in *Table 3*, one can see that triangle elements have a lower L_2 norm compared to their quadrilateral counterparts. However, this changes towards the bottom of the table. for distorted geometries, quadrilateral meshes work better. When it comes to Dirichlet boundary conditions no matter what the geometry is, triangle elements resulted in a lower L_2 norm compared to quadrilateral ones. This trend continues for higher-order functions.

Another comment is the comparison of the gradient calculating method on the same geometry. Catching a generic trend is not possible for the test cases presented. In theory, it is expected to have the L_2 norm decrease as the method shifts from Green-Gauss to

weighted Least Squares. However, data from *Table 1*, follows the theory near perfect for the first three cases, for the last four cases L_2 first decreases then increases. This continues for the second table as well. However, from the third table to the last table for the Neumann condition, the L_2 norm follows the theory. Regarding the Dirichlet condition, the "first increase and then decrease" trend is presented for all tables indifferent to the input equation type. Yet for the first case, the trend follows a decreasing pattern, for the cases from two to seven it follows the trend "first decrease then increase". So it is not wrong to say that as geometry gets more complex Least Squares method without a weighting factor causes less error, yet this result may not be one hundred percent correct since even though the norms are lower in Green Gauss compared to Least Squares, they are still pretty high to run a healthy CFD simulation.

Adding one more comment about the L_2 norm can be that the norm also increases as the functions' nonlinearity increases. One can see this trend by comparing the norm values for the same geometry on different tables.

For test case six and case seven outer dimensions and the circle at the middle are the same, further in both the regions where lies between the outer boundary and the middle circle are meshed via quadrilateral. Due to these similarities, their L_2 error norm for quad elements is pretty much the same. This trend remains unchanged for both Dirichlet and Neumann boundary conditions.

The last comment is, that in structured mesh, test case four, L_2 error norm, changes slightly in the case of Neumann-type boundary conditions, but for Dirichlet-type boundary conditions it will yield the same values regardless of what the boundary functions are.

8 References

- Ali Karakuş Github Repo for $m^{\ell f_\nu} m$ - [link](#)
- Ali Karakuş Lecture Notes for ME485 - [link](#)