Report: Gradient Reconstruction Methods in Fluid Dynamics

Hasan Kaan Özen Batuhan Çoruhlu

November 24, 2023

Contents

1 Introduction **2**

2 Implementation Details **2**
2.1 Function: `greenGaussCell` 2
2.2 Function: `correctGrad` 2
2.3 Function: `greenGaussNode` 3
2.4 Function: `leastSquares` 3
2.5 Function: `weightedLeastSquares` 4

3 Results and Analysis **4**
3.1 Computing Errors 4
3.2 Comparison of Errors 5
3.3 Comparison of Gradient Fields 6
3.4 Different Initial Fields 11

4 Discussion and Conclusion **12**
4.1 Tables 12
4.2 Plots 12
4.3 Conclusion of Plots and Tables 12

1 Introduction

This report uses the *mefvm* package to examine methods for computing fluid property changes. Evaluating various approaches across different mesh topologies, including triangles, quads, and structured/unstructured meshes, is the primary focus. Five methods, "Green-Gauss Cell-Based", "Corrected Green-Gauss Cell-Based", "Green-Gauss Node-Based", "Least-Squares", and "Weighted Least-Squares" methods are implemented.

The accuracy of the methods are tested by computing the infinity norm and mean norm of absolute error between the simulation result and the exact solution computed with the initial field. The results are presented with gradient plots and tables.

2 Implementation Details

In this section, each method will be summarized by the purpose and equations used. The details of every method are presented in the 4.3 section where the codes of methods are provided.

2.1 Function: `greenGaussCell`

Purpose: This function computes the gradient using the Green-Gauss Cell-Based method.

Description:

The Green-Gauss Cell Based method uses the face flux values to gather gradient information on elements sharing the same face. After acquiring the face flux and normal vector, the gradient is computed through Gauss's Divergence Theorem.

2.2 Function: `correctGrad`

Purpose: The `correctGrad` function computes the gradient using the Green-Gauss Cell-Based method with a midpoint correction technique.

Description:

Since the Green-Gauss Cell Based method is only second-order accurate for fully orthogonal grids, a correction process has to be implemented. This function uses the Midpoint Correction, which calculates an auxiliary face flux on the midpoint of two neighboring element centers. Afterward, the face flux is computed using the auxiliary face flux and the element gradients that were calculated on the actual Green-Gauss Cell Based method.

In the correction stage, the function performs specific steps, starting from step 3 outlined in the lecture notes:

1. **Starting Iteration:**

- Calculate face-flux on the midpoint as in Equation 1.

$$\phi'_f = \frac{\phi_e + \phi_f}{2} \tag{1}$$

- Update ϕ_f using Equation 2

$$\phi_f = \phi'_f + \frac{1}{2}\left[(\nabla\phi)_e + (\nabla\phi)_{ef}\right] \cdot \left[\mathbf{r}_f - 0.5(\mathbf{r}_e + \mathbf{r}_{ef})\right] \tag{2}$$

- Revise $\nabla\phi_e$ as in Equation 3

$$\nabla\phi_e = \frac{1}{V_e}\sum_{f=1}^{N_f} \phi_f \cdot nS_f \tag{3}$$

- This update process iterates only twice for convergence purposes.

2.3 Function: `greenGaussNode`

Purpose: The `greenGaussNode` function computes gradients using the Green-Gauss Node-Based method. The method uses nodal flux values to compute face flux and then apply Gauss Divergence Theorem to calculate gradients on element centers.

Description:
The method uses the function "cell2node", implemented in the code to gather nodal information. Then, computing over every face, flux information on vertices sharing the same face are acquired. The flux values are averaged to get the face flux, which will be used in the Gauss Divergence Theorem to calculate gradients on the elements sharing the face.

2.4 Function: `leastSquares`

Purpose: The `leastSquares` function aims to compute fluid dynamics gradients through a least-squares approach. This function utilizes geometric relationships and a matrix solution to derive gradients from neighboring cell data.

Description:

The provided code demonstrates the implementation of a least-squares method for computing gradients in fluid dynamics. It involves a matrix-based calculation by considering a primary cell (e_0) and its neighboring cells (e_i). The function constructs a linear system based on the difference in center coordinates and element fluxes between e_0 and its neighbors.

The calculations start by identifying the differences in center coordinates and element flux values between e_0 and its neighboring cells, utilizing second-order accuracy formulations. A set of stencils involving adjacent cells (e_0, e_1, e_2, and e_3) is employed to compute gradients in a component-wise manner. An example linear system is provided in Equation 4 for a 2D mesh.

$$\underbrace{\begin{bmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \\ x_3 - x_0 & y_3 - y_0 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \partial\phi/\partial x \\ \partial\phi/\partial y \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} \phi_{e1} - \phi_{e0} \\ \phi_{e2} - \phi_{e0} \\ \phi_{e3} - \phi_{e0} \end{bmatrix}}_{\mathbf{b}} \tag{4}$$

The resulting system is overdetermined. Consequently, the system is solved by the linear Least Squares Method. This method is used for systems without an exact solution, consequently approximating a solution by minimizing the squares of the residuals. The solution for $\mathbf{x}$ with this method is presented in Equation 5.

$$\mathbf{x} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{5}$$

2.5 Function: `weightedLeastSquares`

Purpose: The `weightedLeastSquares` function extends the least-squares gradient computation method in fluid dynamics by introducing weighted factors to refine gradient estimations.

Description:

This function expands upon the `leastSquares` method by introducing weighted factors for neighboring cell contributions. Similar to the former method, it employs a least-squares approach but emphasizes the influence of nearby cells by introducing weighted schemes in gradient computations. The least squares method computes the gradient, assuming all neighbors of e_0 are equally weighted, resulting in decreased accuracy.

The code creates a diagonal matrix of using the weight information of the neighboring elements. Afterwards, updates Equation 4 by multiplying each side with the weight matrix W, which can be seen in Equation 6. The solution for the system is presented on Equation 7.

$$\underbrace{\begin{bmatrix} w_1 & 0 & 0 \\ 0 & w_2 & 0 \\ 0 & 0 & w_3 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \\ x_3 - x_0 & y_3 - y_0 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \partial\phi/\partial x \\ \partial\phi/\partial y \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} w_1 & 0 & 0 \\ 0 & w_2 & 0 \\ 0 & 0 & w_3 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} \phi_{e1} - \phi_{e0} \\ \phi_{e2} - \phi_{e0} \\ \phi_{e3} - \phi_{e0} \end{bmatrix}}_{\mathbf{b}} \tag{6}$$

$$\mathbf{x} = \left(\mathbf{W}\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{W}\mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{7}$$

3 Results and Analysis

3.1 Computing Errors

In order to compute errors for the simulation results, an exact solution had to be computed. To do so, the gradient of the initial field was calculated where the initial field and its gradient are provided in Equation 8. Afterward, the coordinates of element centers are multiplied by the gradient to create an exact solution for the gradient.

$$Q = x^2 + y^2, \quad \nabla Q = \begin{pmatrix} 2x \\ 2y \end{pmatrix} \tag{8}$$

The infinity norm of absolute error and average norm of absolute error are calculated as in Equation 9.

$$\begin{aligned} \|\text{Error}\|_{\infty} &= \max\left(|\nabla Q_{\text{simulation}} - \nabla Q_{\text{exact}}|\right), \\ \|\text{Error}\|_{\text{avg}} &= \frac{1}{N}\sum_{i=1}^{N} |\nabla Q_{\text{simulation}} - \nabla Q_{\text{exact}}| \end{aligned} \tag{9}$$

The calculated errors will play an important role in assessing the performance of the methods for different mesh geometries and initial fields.

3.2 Comparison of Errors

Here, the errors will be presented in tables for five different mesh files named, "Cavity Quad Mesh" which is a structured quadratic mesh, "Cavity Triangular Mesh" which is a structured triangular mesh, "Text Mesh" which is an unstructured triangular mesh, "Mixed Mesh" which is a mixed unstructured mesh and "Heat Mesh" which is also a structured mesh.

Table 1: Results for the CavityQuad.msh.

	METHOD				
CavityQuad Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	11.7448979592163	11.7448979592163	1.0204081632659	0.0408163265307282	0.0408163265307284
Average Norm of Error	0.302582257392757	0.325630577673092	0.02141845285031305	0.00166597251145757	0.00166597251146015

Table 2: Results for the CavityTri.msh.

	METHOD				
CavityTri Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	26.769428473147	26.7171592221807	1.30393386448644	0.0694025003318317	0.0694025003318319
Average Norm of Error	0.345263668710109	0.3373873944827309	0.0215564848450563	0.00196498899469132	0.002026292842891014

Table 3: Results for the Heat.msh.

	METHOD				
Heat Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	11.7448979592163	11.7448979592163	1.0204081632659	0.0408163265307282	0.0408163265307284
Average Norm of Error	0.312682215743445	0.335736061131779	0.0211551963751082 4	0.00166597251145757	0.002026292842891014

Table 4: Results for the Text.msh.

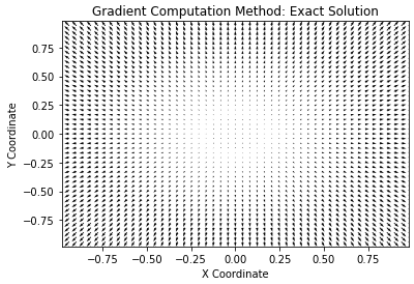
	METHOD				
Text Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	3.28668113108556	3.21654754180171	0.435531730493342	0.380998568548608	0.380998568548608
Average Norm of Error	0.395203275895563	0.410447266482617	0.136650402714696	0.112930241514362	0.114648893142012

Table 5: Results for the Mixed.msh.

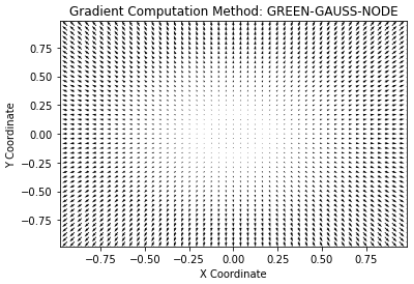
	METHOD				
Mixed Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	59.9786922268329	493.587768867057	11.6116574039991	2.646020828886822	2.646020828886822
Average Norm of Error	4.29594717502539	141.554110591261	1.3121795919842	0.394949983669777	0.425143307152507

3.3 Comparison of Gradient Fields

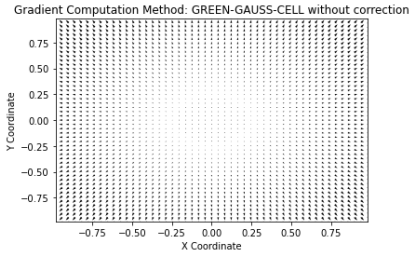
Here, the gradient plots of the exact solution and methods for the aforementioned meshes will be provided.



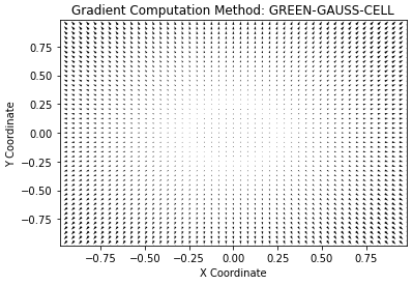
(a) Exact Solution Gradient for Cavity Quad Mesh



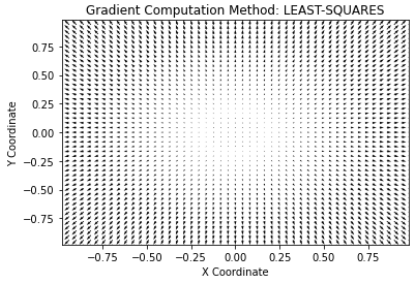
(b) Green-Gauss-Node Based Solution Gradient for Cavity Quad Mesh



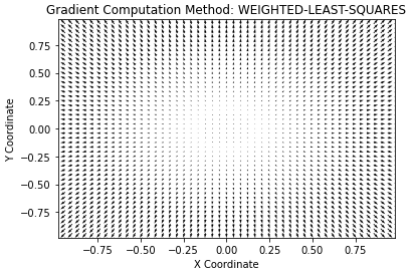
(c) Green-Gauss-Cell Based Solution Gradient for Cavity Quad Mesh



(d) Corrected Green-Gauss-Cell Based Solution Gradient for Cavity Quad Mesh

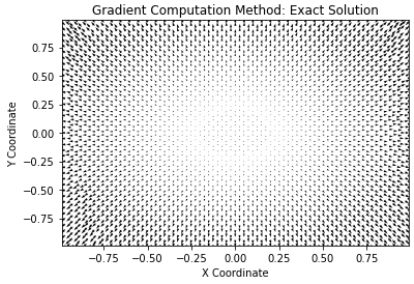


(e) Least Squares Gradient for Cavity Quad Mesh

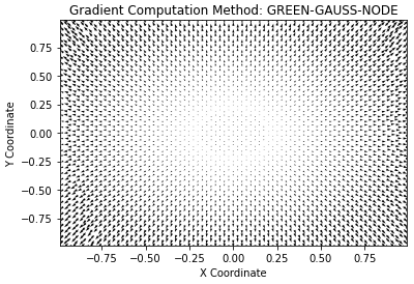


(f) Weighted Least Squares Gradient for Cavity Quad Mesh

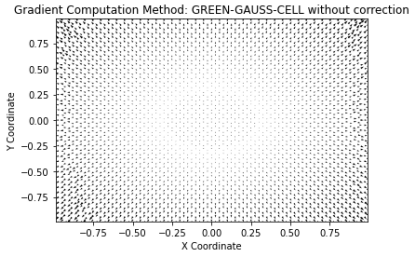
Figure 1: Gradient Plots for Cavity Quad Mesh.



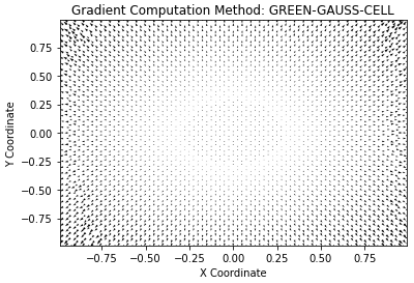
(a) Exact Solution Gradient for Cavity Triangular Mesh



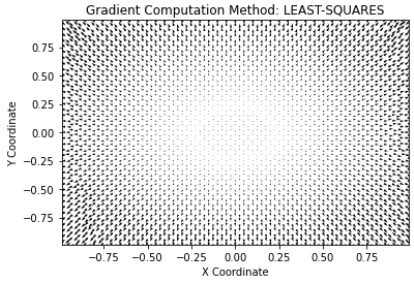
(b) Green-Gauss-Node Based Solution Gradient for Cavity Triangular Mesh



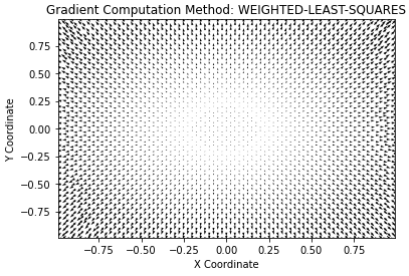
(c) Green-Gauss-Cell Based Solution Gradient for Cavity Triangular Mesh



(d) Corrected Green-Gauss-Cell Based Solution Gradient for Cavity Tri. Mesh

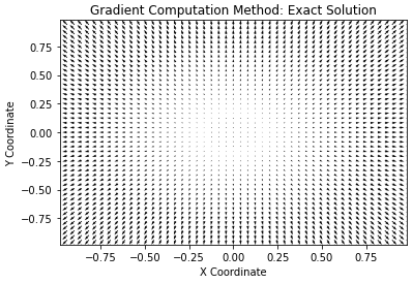


(e) Least Squares Gradient for Cavity Triangular Mesh

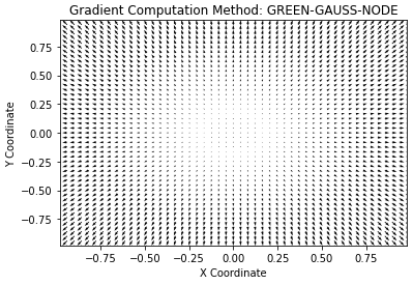


(f) Weighted Least Squares Gradient for Cavity Triangular Mesh

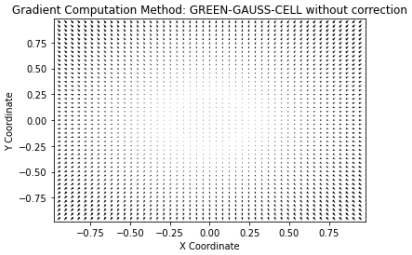
Figure 2: Gradient Plots for Cavity Triangular Mesh.



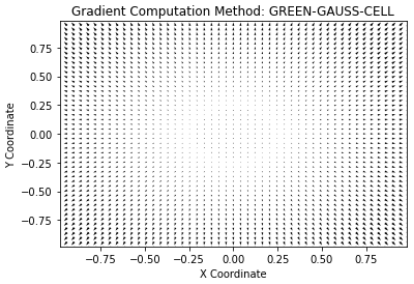
(a) Exact Solution Gradient for Heat Mesh



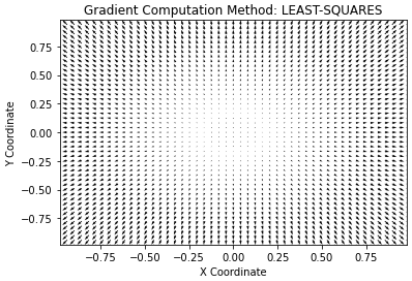
(b) Green-Gauss-Node Based Solution Gradient for Heat Mesh



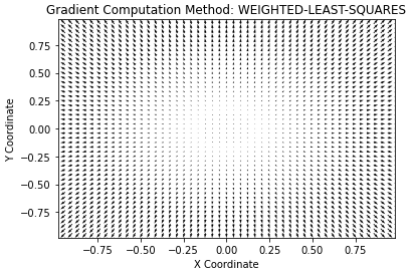
(c) Green-Gauss-Cell Based Solution Gradient for Heat Mesh



(d) Corrected Green-Gauss-Cell Based Solution Gradient for Heat Mesh

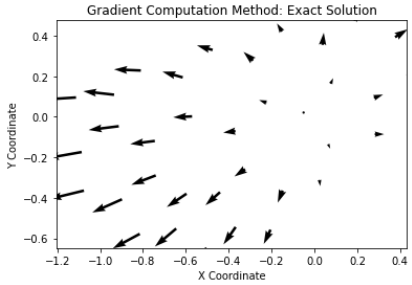


(e) Least Squares Gradient for Heat Mesh

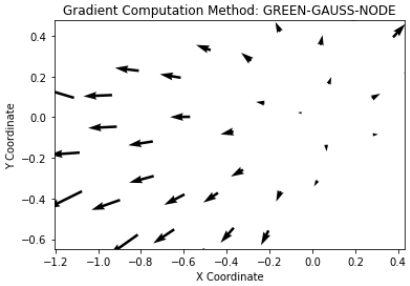


(f) Weighted Least Squares Gradient for Heat Mesh

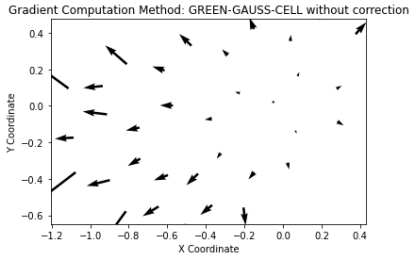
Figure 3: Gradient Plots for Heat Mesh.



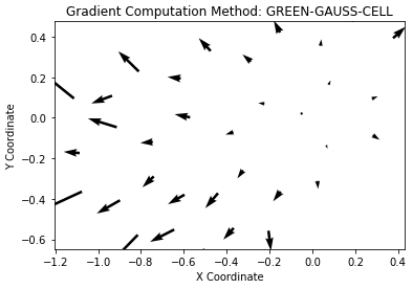
(a) Exact Solution Gradient for Text Mesh



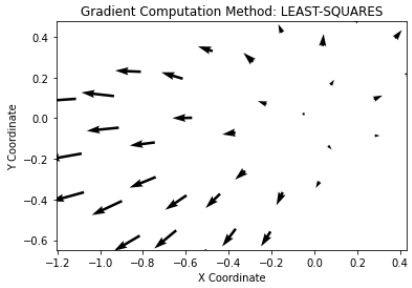
(b) Green-Gauss-Node Based Solution Gradient for Text Mesh



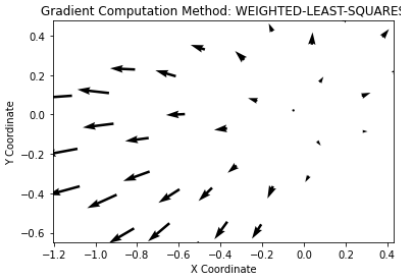
(c) Green-Gauss-Cell Based Solution Gradient for Text Mesh



(d) Corrected Green-Gauss-Cell Based Solution Gradient for Text Mesh

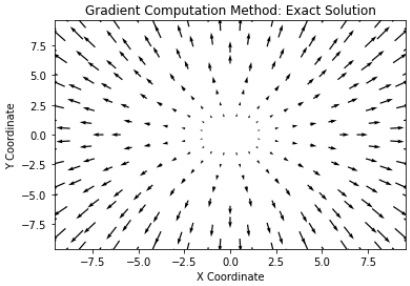


(e) Least Squares Gradient for Text Mesh

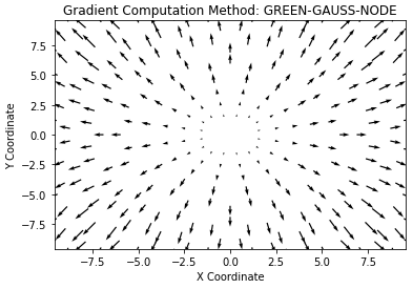


(f) Weighted Least Squares Gradient for Text Mesh

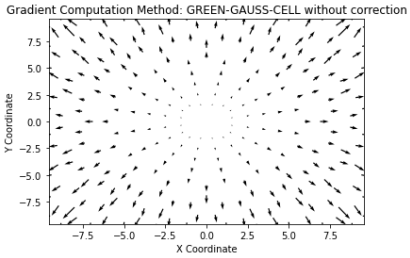
Figure 4: Gradient Plots for Text Mesh.



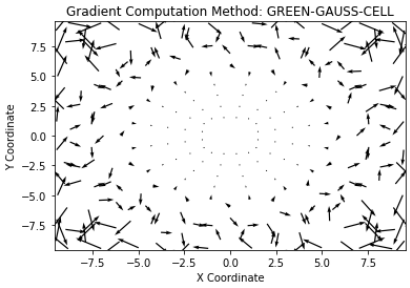
(a) Exact Solution Gradient for Mixed Mesh



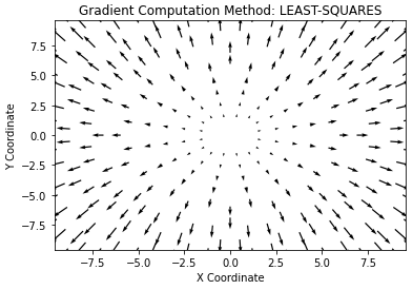
(b) Green-Gauss-Node Based Solution Gradient for Mixed Mesh



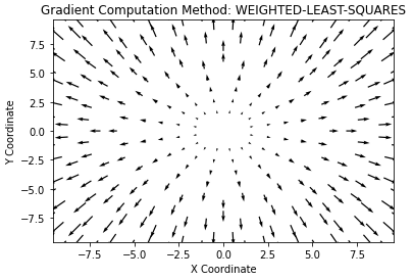
(c) Green-Gauss-Cell Based Solution Gradient for Mixed Mesh



(d) Corrected Green-Gauss-Cell Based Solution Gradient for Mixed Mesh



(e) Least Squares Gradient for Mixed Mesh



(f) Weighted Least Squares Gradient for Mixed Mesh

Figure 5: Gradient Plots for Mixed Mesh.

3.4 Different Initial Fields

In this section, results for different initial fields such as a lower and higher-order polynomial and a trigonometric function will be presented. The fields were initiated on Mixed mesh since it proves to be the most complex mesh. The initiated fields are presented in Equation 10.

$$\begin{aligned} Q_1 &= x^2 + y, & \nabla Q_1 &= (2x, 1). \\ Q_2 &= x^3 + y^2, & \nabla Q_2 &= (3x^2, 2y), \\ Q_3 &= \sin(x) + \cos(y), & \nabla Q_3 &= (\cos(x), -\sin(y)) \end{aligned} \tag{10}$$

Table 6: Results for the Mixed mesh with lower order polynomial boundary and initial fields.

	METHOD				
Mixed Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	44.5625727059686	357.43982587029	7.70207822423588	2.18001792231779	2.18001792231778
Average Norm of Error	2.63887848528627	79.2259692725516	0.886627614994606	0.364905789532696	0.370967017662831

Table 7: Results for the Mixed mesh with higher order polynomial boundary and initial fields.

	METHOD				
Mixed Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	407.502998701512	2794.02526699708	10.4550914952835	33.9832156115266	33.9832156115266
Average Norm of Error	23.6980538608098	548.664146100535	92.1538910058593	5.53253740213086	5.58188639588047

Table 8: Results for the Mixed mesh with trigonometric boundary and initial fields.

	METHOD				
Mixed Mesh	Green-Gauss-Cell	Corrected-Green-Gauss-Cell	Green-Gauss-Node	Least Squares	Weighted Least Squares
Infinity Norm of Error	0.874740755092868	6.354796979624583	0.858689273220048	1.17862950646355	1.17862950646355
Average Norm of Error	0.190047399253083	1.55314036852644	0.250013980088373	0.187890888606593	0.192249479747117

4 Discussion and Conclusion

In this section, the tables and plots provided in Section 3 will be discussed. The key takeaways from the results are:

4.1 Tables

- The infinity norm of error of both normal and corrected Green-Gauss-Cell Based Method are high. This is an unlikely result since the meshes also contain structured grids such as Cavity Quad Mesh. A coding mistake or computation on the boundary might be the problem.
- Comparing Table 1 and 2, structured quadratic mesh provides higher accuracy and precision as both the infinity norm and average norm of errors are lower than the errors of the triangular mesh.
- Observing Table 5, it is evident that the Least-Squares and Weighted-Least-Squares methods provide higher accuracy and precision compared to the Green-Gauss-Node based method in the most complicated mesh of the group which is the Mixed mesh. Moreover, comparing Table 5 with the other tables, the effect of the Weighted-Least-Squares method shines more on this complicated mesh where errors due to weight factors may occur, whereas on other meshes the changes are not significant.

4.2 Plots

- In addition to the points made in Subsection 4.1, observing Figure 5 it is evident that the Corrected-Green-Gauss-Cell Based Methods has computation errors even when compared with the original Green-Gauss-Cell Based Method. Either the code should be revised or the midpoint method does not work for unstructured grids.
- Figure 4 strengthens the theory that both the Weighted-Least-Squares Method and Least-Squares-Method show gradient fields significantly more similar to the gradient field of the exact solution.
- For structured grids, it is hard to observe the differences of the gradient fields of different methods.

4.3 Conclusion of Plots and Tables

It is evident that both the weighted least squares and least squares methods provide more accurate and precise results for complex grids. However, for structured grids, the difference between other methods, assuming Corrected Green Gauss has a mistake, is relatively small and probably will not be worth the computational cost of solving an overdetermined linear system of equations. The key takeaway of this section is to suggest the usage of LSQ and WLSQ method

for unstructured-mixed grids and use the Green-Gauss-Node Based Method for structured grids.

Appendices

Function: `greenGaussCell`

```
def greenGaussCell(self, Qe, Qb):
    msh = self.mesh
    Nfields = Qe.shape[1]
    gradQ = np.zeros((msh.Nelements, Nfields, msh.dim), float)
    self.QF = np.zeros((msh.NFaces, Nfields), float)
    bcid = 0
    for fM, info in msh.Face.items():
        eM = info['owner']; eP = info['neigh']
        qM =  Qe[eM]; qP =  Qe[eP]
        bc = info['boundary']
        normal = info['normal']
        weight = info['weight']
        area = info['area']
        qf = 0.0
        if(self.correct != 'FALSE'):
            weight = 0.5
        if(bc != 0):
            qb = Qb[info['bcid']]
            qP = qb/(1.0-weight) - weight*qM
            qf = weight*qM + (1.0 - weight)*qP
            gradQ[eM, :, 0] = gradQ[eM, :, 0] + qf[:]*area*normal[0]
            gradQ[eM, :, 1] = gradQ[eM, :, 1] + qf[:]*area*normal[1]
            if(msh.dim == 3):
                gradQ[eM, :, 2] = gradQ[eM, :, 2] + qf[:]*area*normal[2]
        else:
            qf = weight*qM + (1.0 - weight)*qP
            gradQ[eM,:, 0] = gradQ[eM, :, 0] + qf*area*normal[0]
            gradQ[eM,:, 1] = gradQ[eM, :, 1] + qf*area*normal[1]
            if(msh.dim == 3):
                gradQ[eM, :, 2] = gradQ[eM, :, 2] + qf[:]*area*normal[2]
            gradQ[eP,:, 0] = gradQ[eP, :, 0] - qf[:]*area*normal[0]
            gradQ[eP,:, 1] = gradQ[eP, :, 1] - qf[:]*area*normal[1]
            if(msh.dim == 3):
                gradQ[eP, :, 2] = gradQ[eP, :, 2] - qf[:]*area*normal[2]
    for eM in msh.Element.keys():
        vol = msh.Element[eM]['volume']
        gradQ[eM] = gradQ[eM]/vol
    return gradQ
```

Function: `correctGrad`

```
def correctGrad(self, Qe, Qb, gQ):
    msh = self.mesh
    Nfields     = Qe.shape[1]
    gradQ       = np.zeros((msh.Nelements, Nfields, msh.dim), float)

   # Fill the rest of this function
    self.QF     = np.zeros((msh.NFaces, Nfields), float)
    bcid = 0
    for fM, info in msh.Face.items():
        eM = info['owner']; eP = info['neigh']
        qM =  Qe[eM]; qP =  Qe[eP]
        print(eM)
        bc      = info['boundary']
        normal = info['normal']
        weight = info['weight']
        area    = info['area']
        xM = msh.Element[eM]['ecenter']
        #get cell center coordinates of neighbor cell
        xP = msh.Element[eP]['ecenter']
        # center coordinates of the face
        xF = info['center']
        qf = 0.0
        # print(xM,xP,eM,fM,msh.Nelements)
        if(self.correct != 'FALSE'):
            weight = 0.5
        #Start Correction using Midpoint, Step 2 is already computed as gQ
        # integrate boundary faces
        if(bc != 0):
            qb              = Qb[info['bcid']]
            qP = qb/(1.0-weight) - weight*qM
            qfprime = weight*qM + (1.0 - weight)*qP#Step 1
            qf= qfprime + (0.5)*np.dot((gQ[eM, :, :]),(xF[:2]-(0.5*(xM[:2]+x
            gradQ[eM, :, 0] = gQ[eM, :, 0]#Step 3.b
            gradQ[eM, :, 1] = gQ[eM, :, 1]#Step 3.b
            if(msh.dim == 3):#Step 3.b
                gradQ[eM, :, 2] = gQ[eM, :, 2] + qf[:]*area*normal[2]#Step 3
        # integrate internal faces
        else:
            qfprime = weight*qM + (1.0 - weight)*qP#Step 1
            qf= qfprime + (0.5)*np.dot(((gQ[eM, :, :])+gQ[eP, :, :]),(xF[:2]-
            gradQ[eM,:, 0] = gQ[eM, :, 0] + qf*area*normal[0]#Step 3.b
            gradQ[eM,:, 1] = gQ[eM, :, 1] + qf*area*normal[1]#Step 3.b
            if(msh.dim == 3):#Step 3.b
                gradQ[eM, :, 2] = gQ[eM, :, 2] + qf[:]*area*normal[2]#Step 3

            gradQ[eP,:, 0] = gQ[eP, :  , 0] - qf[:]*area*normal[0]#Step 3.b
```

```
                gradQ[eP,:, 1] = gQ[eP, :, 1] - qf[:]*area*normal[1]#Step 3.b
                if(msh.dim == 3):#Step 3.b
                    gradQ[eP, :, 2] = gQ[eP, :, 2] - qf[:]*area*normal[2]#Step 3
        return gradQ
```

Function: `greenGaussNode`

```
    def greenGaussNode(self, Qe, Qb):
            msh = self.mesh
            Nfields = Qe.shape[1]
            gradQ = np.zeros((msh.Nelements, Nfields, msh.dim), float)
            QF = np.zeros((msh.NFaces, Nfields), float)
            Qv = msh.cell2Node(Qe, Qb, 'average')

            # Fill the rest of this function
            for fM, info in msh.Face.items():
                eM = info['owner']; eP = info['neigh']
                bc     = info['boundary']
                normal = info['normal']
                area   = info['area']
                nodes = info['nodes']#Get node info from face
                QF[fM] = (Qv[nodes[0]]+Qv[nodes[1]])*0.5#Get face value through
                #Calcualte gradient through face value and normal
                if(bc != 0):#Check boundary
                    qb                = Qb[info['bcid']]
                    QF[fM] = qb
                    gradQ[eM,:,0] = gradQ[eM,:,0] + QF[fM]*area*normal[0]
                    gradQ[eM,:,1] = gradQ[eM,:,1] + QF[fM]*area*normal[1]

                else:
                    gradQ[eM,:,0] = gradQ[eM,:,0] + QF[fM]*area*normal[0]
                    gradQ[eM,:,1] = gradQ[eM,:,1] + QF[fM]*area*normal[1]

                    gradQ[eP,:,0] = gradQ[eP,:,0] - QF[fM]*area*normal[0]
                    gradQ[eP,:,1] = gradQ[eP,:,1] - QF[fM]*area*normal[1]

            for eM in msh.Element.keys():
                vol = msh.Element[eM]['volume']
                gradQ[eM] = gradQ[eM]/vol
            return gradQ
```

Function: `leastSquares`

```
    def leastSquares(self, Qe, Qb):
```

```
        msh = self.mesh
        Nfields = Qe.shape[1]
        gradQ = np.zeros((msh.Nelements, Nfields, msh.dim), float)
        for elm, info in msh.Element.items():
            neigh = info['neighElement']#Get neignbor data
            sizeneigh = len(neigh)#Get number of neighbors
            bc = info['boundary']#Get boundary data
            A = np.zeros((sizeneigh, msh.dim))
            b = np.zeros(sizeneigh)
            for i in range (sizeneigh): #compute over the neigbors of the eleme
                Nneigh = neigh[i]#get neighbord id
                qM =  Qe[elm]#get q data for E
                qP =  Qe[Nneigh]#get q data for N
                xE = msh.Element[elm]['ecenter']#get neigh coordinate
                xP = msh.Element[Nneigh]['ecenter']#get center coordinate
                A[i,:] = (xP-xE)[:msh.dim]#construct A matrix
                b[i] = qP - qM #construct B matrix
            AtpA=np.dot(A.T,A)#construct Atranspose.A
            Atpb=np.dot(A.T,b)#construct Atranspose.b
            x=np.linalg.solve(AtpA,Atpb)#solver for x
            gradQ[elm,:]=x#store gradQ
        return gradQ
```

Function: `weightedLeastSquares`

```
    def weightedLeastSquares(self, Qe, Qb):
        msh = self.mesh
        Nfields = Qe.shape[1]
        gradQ = np.zeros((msh.Nelements, Nfields, msh.dim), float)
        for elm, info in msh.Element.items():
            neigh = info['neighElement']#Get neignbor data
            sizeneigh = len(neigh)#Get number of neighbors
            bc = info['boundary']#Get boundary Data
            weight = info['weight']#Get Weight Data
            A = np.zeros((sizeneigh, msh.dim))
            b = np.zeros(sizeneigh)
            w = weight#store weight data
            w = np.diag(w.flatten())#construct diagonal weight matrix
            for i in range (sizeneigh): #compute over the neigbors of the eleme
                Nneigh = neigh[i]#get neighbord id
                qM =  Qe[elm]#get q data for E
                qP =  Qe[Nneigh]#get q data for N
                xE = msh.Element[elm]['ecenter']#get neigh coordinate
                xP = msh.Element[Nneigh]['ecenter']#get center coordinate
                A[i,:] = (xP-xE)[:msh.dim]#construct A matrix
```

```
            b[i] = qP- qM#construct B matrix
        wA=np.dot(w,A)
        wb=np.dot(w,b)
        AtpA=np.dot(wA.T,wA)#construct Atranspose.A
        Atpb=np.dot(wA.T,wb)#construct Atranspose.b
        x=np.linalg.solve(AtpA,Atpb)#solver for x
        gradQ[elm,:]=x#store gradQ
return gradQ
```