Report: Computing Diffusion with Finite Volume Method For Orthogonal Grids

Hasan Kaan Özen Batuhan Çoruhlu

December 3, 2023

Contents

1 Introduction

This report uses the *mefvm* package to compute the diffusion equation by forming a system of linear equations based on the discredited form of the equation and solving for the unknown scalar properties of the element centers ϕ_e.

The accuracy of the methods are put in comparison with two different quadratic and triangular meshes on the same domain and three different linear solvers and preconditioners. The element size of the meshes are 2400 and 946 respectively.

2 Implementation Details

In this section, the method will be summarized by the purpose and equations used. The details of the method are presented in the 4 section where the codes of methods are provided.

2.1 Function: `Assemble Orthogonal`

Purpose: This function computes the diffusion equation assuming the grid is orthogonal.

Description: The linear system is formed by assuming the directions of the shared face normals and the gradient are collinear. The diffusion equation in Equation 1 is used to form the discrete linear system in Equation 2. In Equations 3, 4 and 5, definitions of the linear system are presented. The linear system A consists of every coefficient related to ϕ and the b matrix consists of the source term, Q_C and every coefficient not related to ϕ such as the coefficient A_b at the boundary, ϕ_b. These known coefficients are denoted as N.

$$-\nabla \cdot (\Gamma^{\circ} \nabla \phi) = Q^{\circ} \tag{1}$$

$$\underbrace{\begin{bmatrix} A_c & A_e & A_w & A_n & A_s \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \phi_c \\ \phi_e \\ \phi_w \\ \phi_n \\ \phi_s \end{bmatrix}}_{\mathbf{q}} = \underbrace{\begin{bmatrix} b_c \end{bmatrix}}_{\mathbf{b}} \tag{2}$$

$$A_C, w = \Gamma^{\circ}_w \frac{S_w}{\delta r_w}, \quad A_W = -\Gamma^{\circ}_w \frac{S_w}{\delta r_w} \tag{3}$$

$$A_C = -(A_E + A_W + A_N + A_S) \tag{4}$$

$$b_C = Q_C V_C - (N_e + N_w + N_n + N_s) \tag{5}$$

2.2 Preconditioners

The preconditioners used in the problem are **Jacobian**, **Incomplete Lu**, and **Algebraic Multigrid** methods. Jacobian precondition uses the diagonal dominance of the system to relax the system. It is a simple and efficient precondition. The Incomplete Lu method estimates the Lu factorization of the matrix for the solution and is more computationally expensive that Jacobi but is more solid. The Algebraic Multigrid method is best suited for complex problems and is the most computationally intensive precondition.

2.3 Solvers

The solvers used in the problem are **Direct**, **Generalized Minimal Residual** and **Conjugate Gradient**. The Direct solver computes the exact solution and is good for high-precision small systems. The Generalize Minimal Residual solver is an iterative solver used for a wide range of systems but has a high memory cost. The Conjugate Gradient solver is also an iterative solver with low memory cost and is good for sparse matrices which is the case on this report.

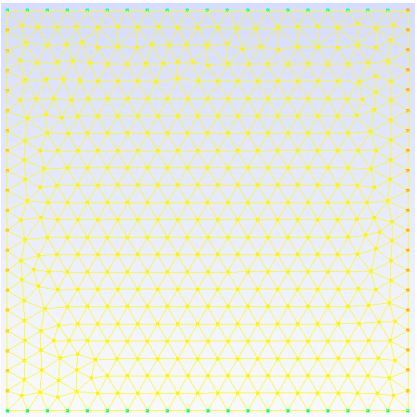
3 Results and Discussion

3.1 Default Setting

The default settings for the system are presented in Table 1. The meshes that will be put in comparison are presented in Figure 2. The results are presented in Tables 2, 3 and 4for all solvers and different preconditionings.

Table 1: Default Settings

Initial Condition	$x + y$
Boundary Condition	$\sin(\pi x)\sin(\pi y)$
Diffusion Coefficient	1
Diffusion Source	$2\pi^2 \sin(\pi x)\sin(\pi y)$
Solver Tolerance	$1.00e-08$



(a) Cavity Triangular Mesh

(b) Cavity Quadratic Mesh.

Figure 1: Pictures of the meshes.

Table 2: Results of solvers with Jacobi Preconditioning

RESULTS						
Meshes	**Cavity Quadratic**			**Cavity Triangular**		
Solvers	Direct	GMRES	CG	Direct	GMRES	CG
Infinity Norm	1.37e-03	1.37e-03	1.37e-03	3.33e-02	3.33e-02	3.33e-02
L2 Norm	1.88e-06	1.88e-06	1.88e-06	1.43e-04	1.43e-04	1.43e-04
Number of Iterations	0	78	44	0	202	112

Table 3: Results of solvers with Incomplete Lu Preconditioning

RESULTS						
Meshes	**Cavity Quadratic**			**Cavity Triangular**		
Solvers	Direct	GMRES	CG	Direct	GMRES	CG
Infinity Norm	1.37e-03	1.37e-03	1.37e-03	3.33e-02	3.33e-02	3.33e-02
L2 Norm	1.88e-06	1.88e-06	1.88e-06	1.43e-04	1.43e-04	1.43e-04
Number of Iterations	0	25	24010	0	3	3

Table 4: Results of solvers with AMG Preconditioning

RESULTS						
Meshes	**Cavity Quadratic**			**Cavity Triangular**		
Solvers	Direct	GMRES	CG	Direct	GMRES	CG
Infinity Norm	1.37e-03	1.37e-03	1.37e-03	3.33e-02	3.33e-02	3.33e-02
L2 Norm	1.88e-06	1.88e-06	1.88e-06	1.43e-04	1.43e-04	1.43e-04
Number of Iterations	0	8	8	0	11	10

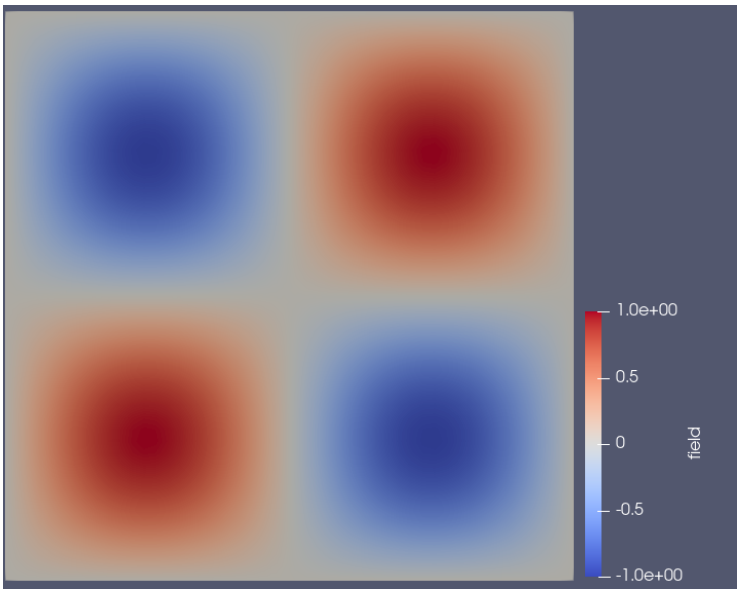


Figure 2: Resulting scalar field.

Observing the results it can be concluded that all combinations result with equal norms. However, for both meshes, the best preconditioning method is AMG as it results in the least amount of iterations. This is because Jacobian preconditioning is a simple relaxation method that results in a higher number of iterations and the Incomplete Lu takes 24000 iterations for CG solver at Cavity Quadratic mesh.

3.2 Quadratic Setting

The complexity of the problem has been changed with the change in the boundary condition from trigonometric to a polynomial. Since the AMG preconditioning is observed to be the feasible preconditioning method, the results of other preconditioning methods are not provided.

Table 5: Complex Settings

Initial Condition	$x + y$
Boundary Condition	$x^2 * y^2)$
Diffusion Coefficient	1
Diffusion Source	$2\pi^2 \sin(\pi x) \sin(\pi y$
Solver Tolerance	$1.00e - 08$

Table 6: Results of solvers with AMG Preconditioning

RESULTS						
Meshes	**Cavity Quadratic**			**Cavity Triangular**		
Solvers	Direct	GMRES	CG	Direct	GMRES	CG
Infinity Norm	1.92	1.92	1.92	3.33	1.79	1.79
L2 Norm	2.26	2.26	2.26	1.43	2.77	2.77
Number of Iterations	0	7	7	0	9	9

It is evident that the accuracy of the system dramatically decreases with the change from periodic conditions to polynomials. However, the current complexity of the setting does not change the precision of the solvers

4 Conclusion

In this report two different types of meshes have been thoroughly tested with different combinations of preconditioners and solvers and different boundary conditions. It has been observed that all combinations lead to the same result probably because of the orthogonality of the mesh. The Algebraic multi-grid preconditioning leads to the least number of iterations because it is best suited for discretized partial differential equations. Different mesh resolutions have not been tested since there is no difference in the number of iterations between different mesh types. The difference in errors is because there is no true orthogonality on the boundary elements, leading to increased error on triangular mesh. It also should be noted that Incomplete Lu preconditioning is not reliable because of the oscillations near the aimed tolerance.

Appendix

```
def assembleOrthogonal(self, Qe, Qb):
    # copy mesh class
    msh = self.mesh
    # Create dummy memory to hold indices and values of the sparse matrix
```

```
# A(i,j) = val, i goes to "rows", j goes to "cols"
# Note that we don't know the exact number of nonzero elements so give a
rows = np.zeros((5*msh.Nelements), int) -1
cols = np.zeros((5*msh.Nelements), int) -1
vals = np.zeros((5*msh.Nelements), float) -1
# RHS vector (b in the homework) is not sparse
rhs  = np.zeros((msh.Nelements,1), float)
# This holds the number of non-zero entries,
# whenever you add an entry increase sk by one, i.e. sk = sk+1
Ab=0
Af=0
sk = 0
ctr = 0
k=self.Ke
Qs=self.Qs
for elm, info in msh.Element.items():
    etype   = info['elementType']
    nfaces  = msh.elmInfo[etype]['nfaces']
    xM = info['ecenter']
    vol = info['volume']#Get Volume Info
    rows[sk] = elm #Assign Ac location
    cols[sk] = elm #Assign Ac location
    vals[sk] = 0 #Assign Ac=0
    sk = sk+1 #Increase entry
    for face in range(nfaces):#Faces of element[elm]
        area = info['area'][face]#Get face area info
        bc = info['boundary'][face]#Get boundary info
        eP = info['neighElement'][face]#Get neighbor element info
        xP = msh.Element[eP]['ecenter']#Get neign elm coords
        Sf = area
        rows[sk] = elm#Assign center info
        cols[sk] = eP#Assign neigh info
        diff = xP-xM#Compute coords difference
        magdiff=np.linalg.norm(diff)
        if(bc!=0):#Drichlett Condition
            xP = info['fcenter'][face]#Coords of boundary face center
            diff = xP-xM#Compute Coords diff btw Boundary and elm center
            magdiff=np.linalg.norm(diff)
            Ab = k[elm]*Sf/magdiff
            vals[sk-face-1] = vals[sk-face-1]+ Ab# Ac = Ac + Ab
            vals[sk] = 0#Assign Af = 0 since boundary
            sk = sk+1
            rhs[elm] = rhs[elm]-(-Ab*Qb[ctr])# b = b - (-Ab*Qb)
            ctr = ctr+1
        else:
            Af = -k[elm]*Sf/magdiff
```

```
                vals[sk] = Af # Assign Af to Af
                vals[sk-face-1] = vals[sk-face-1]- Af# Ac = Ac + (-Af)
                sk = sk+1
        rhs[elm] = -1*((Qs[elm]*vol) + rhs[elm])# -b = -(Qs*Vol + b)
# Delete the unused memory
rows = rows[0:sk]
cols = cols[0:sk]
vals = vals[0:sk]
# Convert (i,j,val) tuple to sparse matrix
A   = sp.sparse.coo_matrix((vals[:], (rows[:], cols[:])),
    shape=(msh.Nelements, msh.Nelements), dtype=float)
return A, rhs;
```