CFD Assignment 2 Convection and diffusion

110033640 Yi-En Chou

December 12, 2021

1 Problem Description

The present task is to provide numerical solution of a steady Convection and diffusion problem. Fora two dimensional convection diffusion equation,

$$\frac{\partial UT}{\partial x} + \frac{\partial VT}{\partial y} = \Gamma(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})$$

Here, for simplicity, U=V=1 and the boundary conditions are shown in the following Figure.

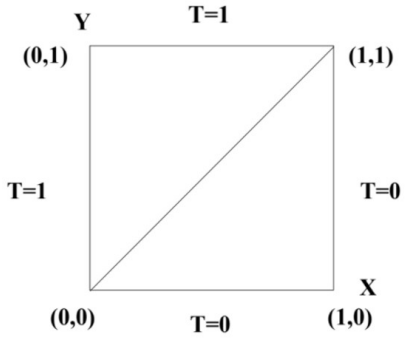

Figure 1: Boundary Condition

2 Introduction of methodology adopted

2.1 Computational method

Instead of Finite Difference Method(FDM), Finite Volume method(FVM) is the computational method implemented in this task, for it being a better approach in the aspect of conservation.

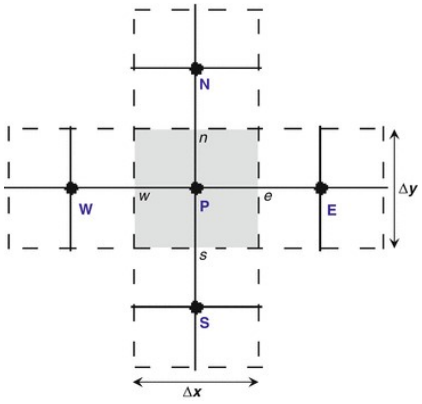

Figure 2: Finite Volume method(FVM)

2.2 Equation Discretization

For U=V=1, the two dimensional convection diffusion equation becomes,

$$\frac{\partial T}{\partial x} + \frac{\partial T}{\partial y} = \Gamma(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})$$

By double integrate the equation, the equation becomes

$$\iint \frac{\partial T}{\partial x} + \frac{\partial T}{\partial y} = \Gamma(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})\, dx\, dy$$

$$\rightarrow u\Delta y T_e - u\Delta y T_w + v\Delta x T_n - v\Delta x = \Gamma\Delta y \frac{\partial T}{\partial x}\bigg|_e - \Gamma\Delta y \frac{\partial T}{\partial x}\bigg|_w + \Gamma\Delta x \frac{\partial T}{\partial y}\bigg|_n - \Gamma\Delta x \frac{\partial T}{\partial y}\bigg|_s$$

Let $\dot{m} = \rho$uA, where $\rho = 1$, u=U=V=1 and A=$\Delta x = \Delta y$, the equation becomes

$$\dot{m}(T_e - T_w + T_n - T_s) = D_e(T_E - T_P) - D_w(T_P - T_W) + D_n(T_N - T_P) - D_s(T_P - T_S)$$

$$D = \frac{\Gamma\Delta y}{\Delta x}$$

In order to generally applied various of schemes, the concept of flux limiter operator ϕ is implemented, where the definition and details will be later address in the next subsection. By so, we substitute terms at cell face as

$$T_e = T_P + \frac{\phi_e}{2}(T_P - T_W)$$

$$T_w = T_W + \frac{\phi_w}{2}(T_W - T_{WW})$$

$$T_n = T_P + \frac{\phi_n}{2}(T_P - T_S)$$

$$T_s = T_S + \frac{\phi_s}{2}(T_S - T_{SS})$$

The equation is then rewritten into standard form,

$$T_P = \frac{A_e T_e + A_w T_w + A_n T_n + A_s T_s + A_{WW} T_{WW} + A_{SS} T_{SS}}{A_P}$$

where

$$A_E = D_e; \quad A_W = 0.5\dot{m}(\phi_e + \phi_w + 1) + D_w$$

$$A_N = D_n; \quad A_S = 0.5\dot{m}(\phi_n + \phi_s + 1) + D_s$$

$$A_{WW} = -0.5\dot{m}\phi_w; \quad A_{SS} = -0.5\dot{m}\phi_s$$

$$A_P = A_E + A_W + A_N + A_S + A_{WW} + A_{SS}$$

2.3 Scheme for cell wall

To more generally apply schemes into our computation, the flux-limiter operator (ϕ) is implemented, where ϕ is the ratio of the slope between the cell wall and its previous node to the slope between its previous node to the node further ahead.

$$\phi_e = 2(\frac{T_e - T_P}{T_P - T_W})$$

We obtain r from the grid value, where r is the ratio of the slope between the following node of the cell wall and its previous node to the slope between its previous node to the node further ahead.

$$r_e = (\frac{T_E - T_P}{T_P - T_W})$$

where ϕ is can be represented as a function of r (e.g $\phi(r)$). By the given definition or design, we are capable of obtaining the value of cell wall from the known information from nodes. Where

Upwind scheme gives,

$$\phi(r) = 0$$

Second order Upwind scheme gives,

$$\phi(r) = 1$$

Central Differencing scheme gives,

$$\phi(r) = r$$

QUICK scheme gives,

$$\phi(r) = \max[0, min(2r, 2, \frac{2(r + |r|)}{r + 3})]$$

minmod scheme gives,

$$\phi(r) = \max[0, min(r, 1)]$$

van Leer scheme gives,

$$\phi(r) = \frac{(r + |r|)}{1 + r}$$

2.4 Convergence Improvement

As for the results to not diverge with iterations, modification is required to be added to. It is the under relaxation that will be implemented in order to deal with the task. For this task, there are two place we apply under relaxation: the updating of velocity and pressure correction, and they are independent from another.

$$u_{n+1} = u_n + \omega_{vel}(u_{n+1} - u_n)$$

$$v_{n+1} = v_n + \omega_{vel}(v_{n+1} - v_n)$$

$$p'_{n+1} = p'_n + \omega_p(p_{n+1} - p_n)$$

For this task, ω_p is set 0.8 for all cases; at Re=100 ω_{vel} is set 0.8 under all schemes; at Re=1000 ω_{vel} is set 0.25 for CDS to converge, and 0.05 for QUICK and MUSCL to converge; at Re=5000 ω_{vel} is set 0.01 for CDS to converge, and QUICK and MUSCL is incapable of convergence.

3 Results and discussions

(Note: The simulation in this section is run under mesh size 161x161)

3.1 Upwind Scheme

This is the result applying the upwind scheme, where the solution does not further approach the exact solution of Pe=∞ after Pe=10.

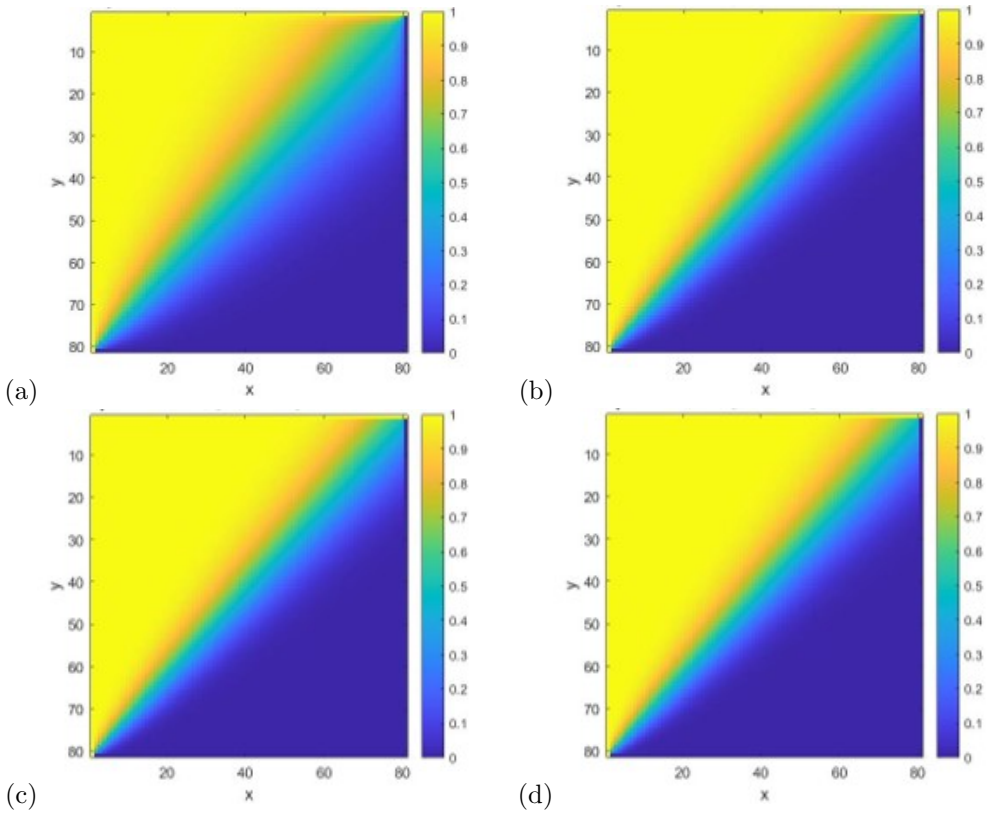

Figure 3: Result of (a) Pe=1 (b) Pe=10 (c) Pe=100 (d) Pe=∞

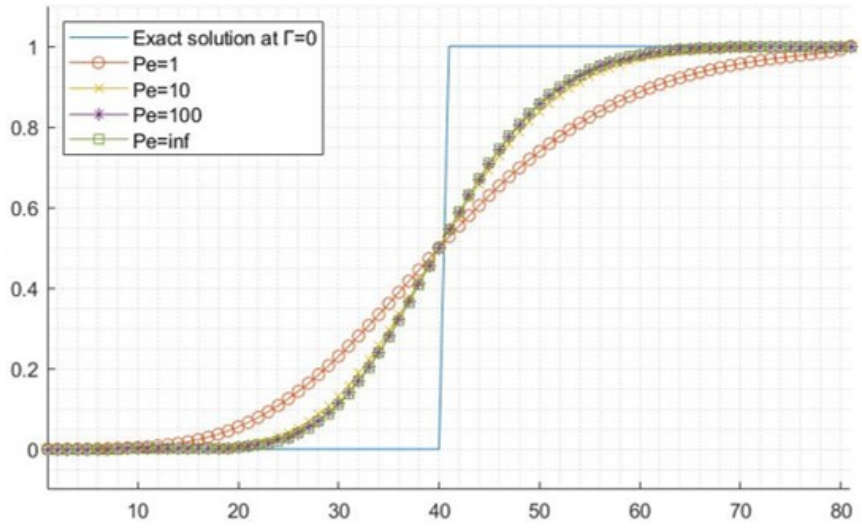

Figure 4: Comparison in predicted result with analytic solution

3.2 CDS

This is the result applying the central differencing scheme, where the span on the diagonal converges with the increase of the peclet number. Note that, the scheme at its original form is highly unstable, therefore, it requires relaxation in order for the deviation to converge. For Pe=10, the weight is set 0.25, and for Pe=100, the weight is set 0.01, and the result for Pe=∞ under this scheme is unreachable.

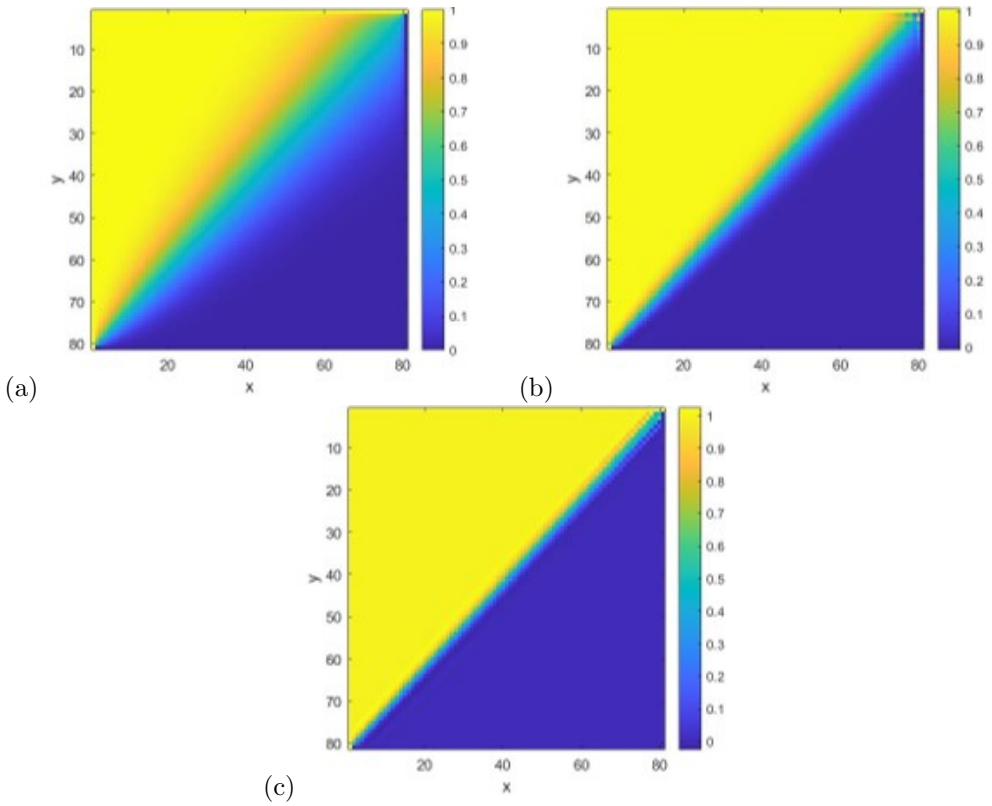

Figure 5: Result of (a) Pe=1 (b) Pe=10 (c) Pe=100

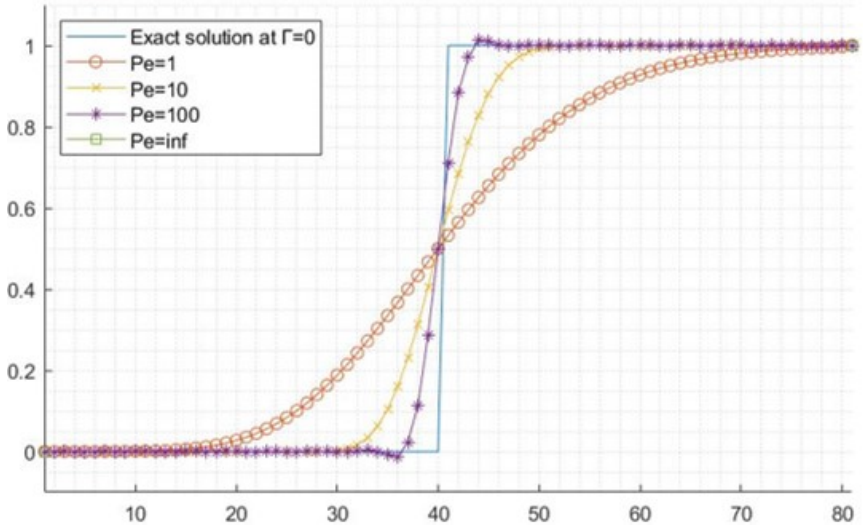

Figure 6: Comparison in predicted result with analytic solution

3.3 QUICK

This is the result applying the QUICK scheme, likewise, the span on the diagonal converges with the increase of the peclet number. With respect to the CDS, the QUICK scheme is of lower accuracy, but it is not as unstable as the CDS, the weight of relaxation at pe=100 is 0.5, which is a lot bigger than that of the CDS, where 0.01 is required, and thus enable us to reach the result for Pe=∞ under this scheme.

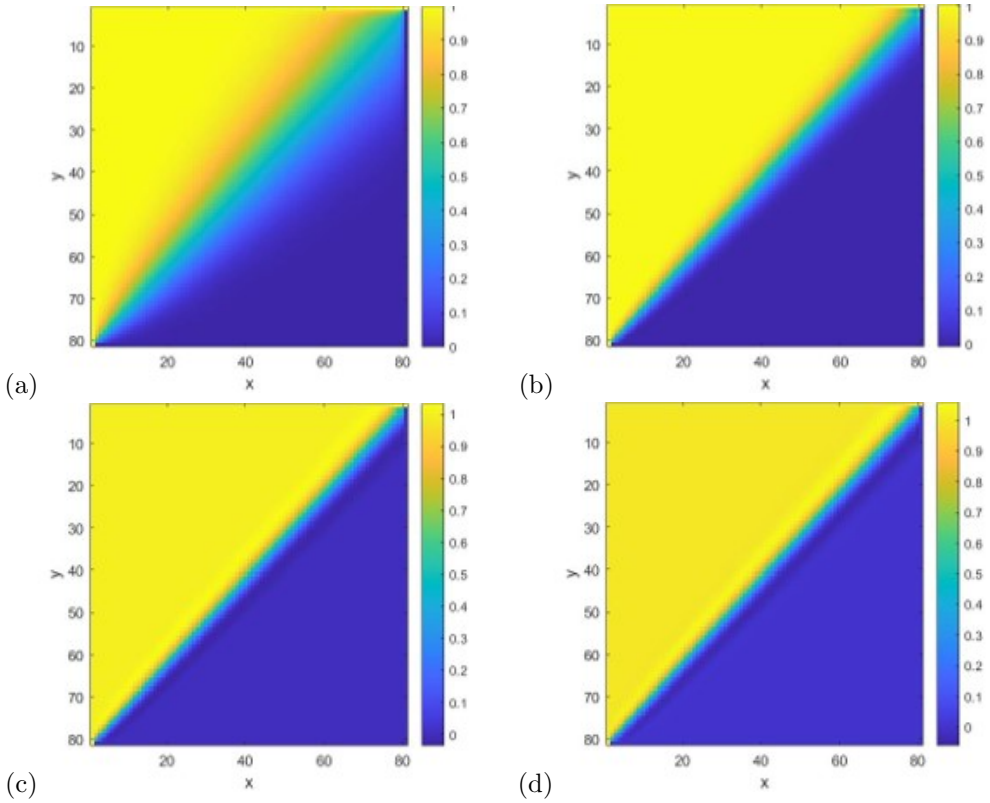

Figure 7: Result of (a) Pe=1 (b) Pe=10 (c) Pe=100 (d) Pe=∞

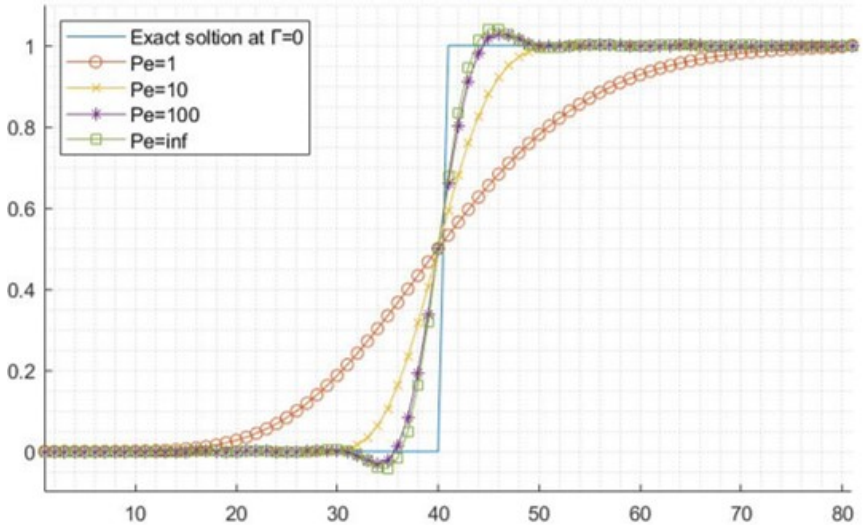

Figure 8: Comparison in predicted result with analytic solution

3.4 minmod

And these are the result applying the minmod scheme, this scheme is a second order TVD scheme, where its span on the diagonal also converges with the increase of the peclet number. Different from the last two schemes, there are no overshoot or undershoot in this scheme.

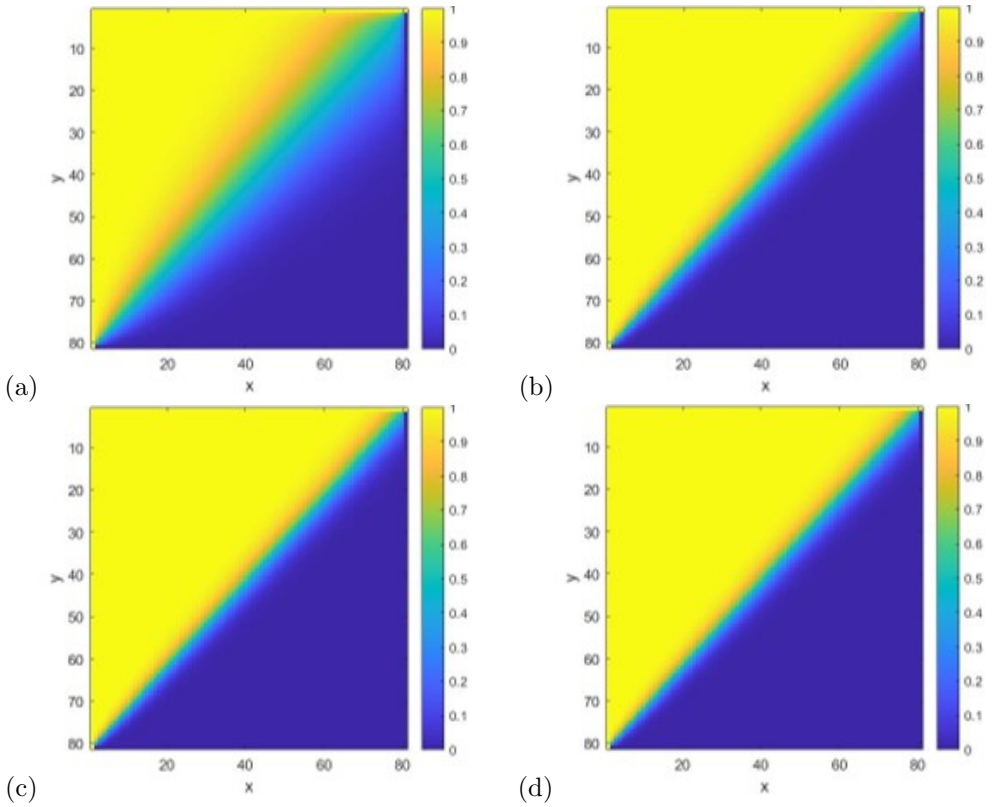

Figure 9: Result of (a) Pe=1 (b) Pe=10 (c) Pe=100 (d) Pe=∞

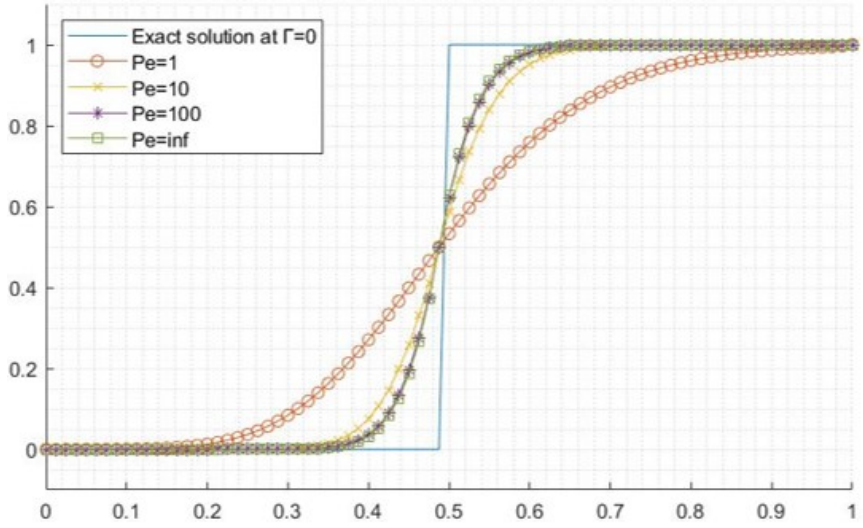

Figure 10: Comparison in predicted result with analytic solution

3.5 Comparison between Schemes

This is solution at Pe=∞ of all the schemes plotted together, and we can see that the quick scheme has the highest accuracy, followed by 2^{nd} order upwind, min-mod then upwind and hybrid. The CDS is unable to converge under this setting, but it is consider an even higher accuracy since its result at Pe=100 is already closer to the exact solution than the result at Pe=∞ under the QUICK scheme.

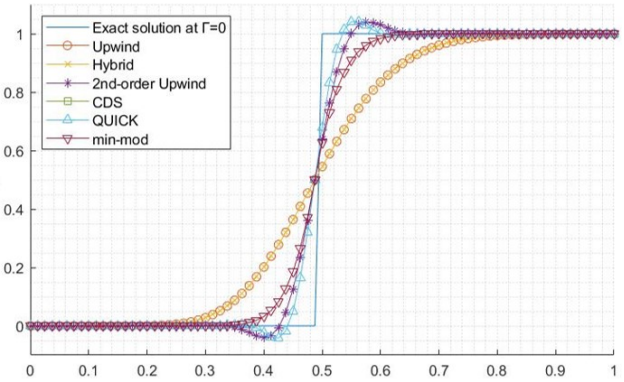

Figure 11: Comparison between Schemes at Pe=∞

3.6 Effect of mesh size on the result

Mesh size also influence the accuracy, with a smaller mesh size, the solutions tend more to the exact solution, we can see that from both the upwind scheme and the quick scheme. However, not unlimitedly, for upwind scheme, the solution under mesh size 321x321 is the same as the solution under 161x161, and for the QUICK scheme, the solution can not even converge under the setting of 321x321.

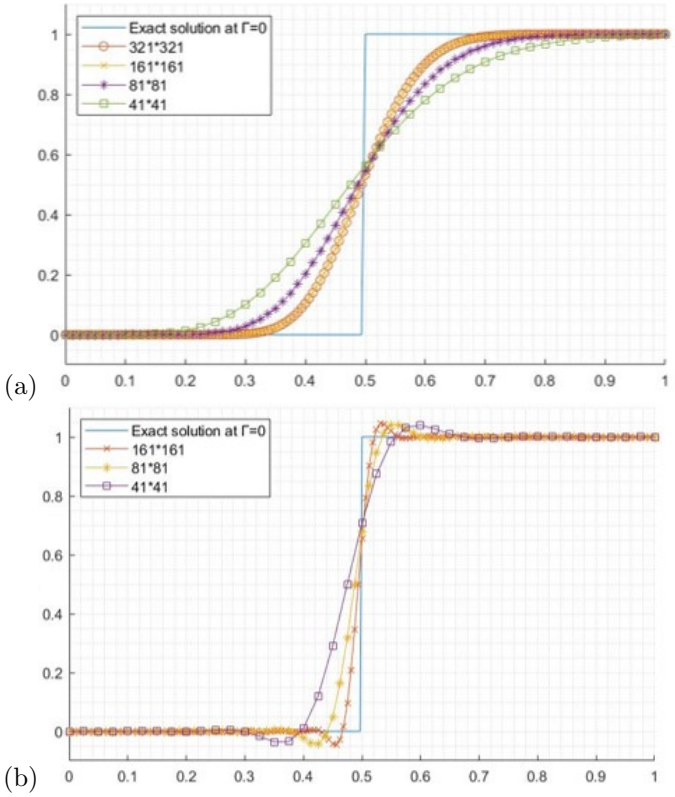

Figure 12: Comparison between mesh size under (a) Upwind (b) QUICK

3.7 Error discussion base on TVD

We can see that the Upwind scheme is of the least accuracy, where it is rational since it's only a first-order TVD scheme, while the other three being bounded or almost bounded within the second order TVD region. The CDS and the QUICK are both not completely bounded within TVD region, where they both obviously results in new maximum and minimum where TVD is not allowed. And as for min-mod, this scheme is completely bounded within the 2nd order TVD region, therefore, there will not be any overshoot or undershoot, but as well get a solution which is of an accuracy of 2nd order. So here we can relates our results with schemes characteristics.

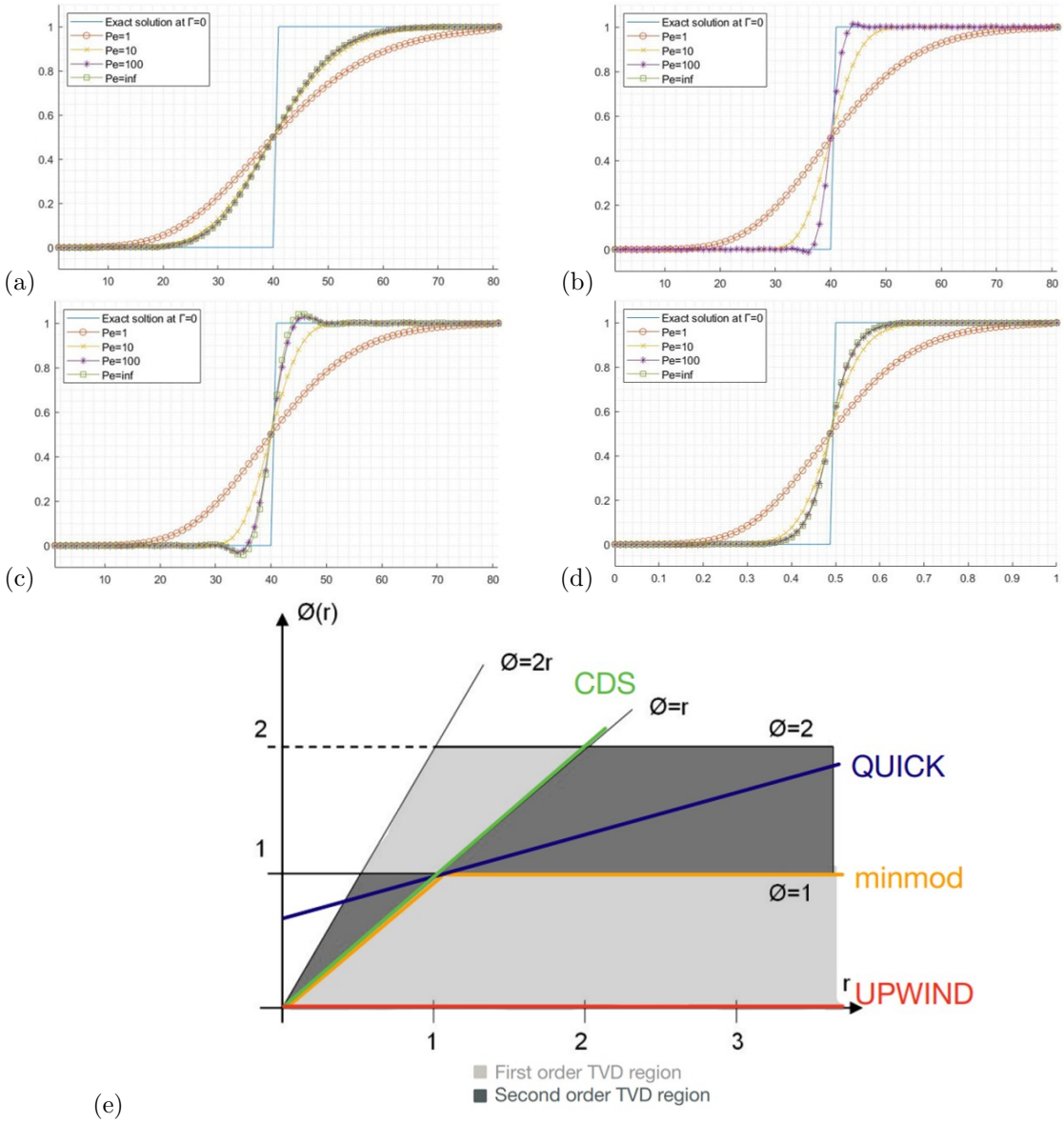

Figure 13: Comparison with analytic solution under (a) Upwind (b) CDS (c) QUICK (d) minmod (e) TVD regions

3.8 Discussion on oscillation suppression

For high order scheme, oscillation is observable when Γ approaches 0. There are two methods found in this task in order to suppress oscillation. First, by flux limiter, or the choice of scheme. The flux limiter limits values into a certain bound, which is decided by the design of schemes, so that the oscillation will therefore be suppressed, yet the magnitude of the artificial diffusion is often enhanced accordingly. From the analysis of TVD, TVD region gives us a convenient view on predicting whether a scheme is

of oscillation, where if the scheme sits within the TVD region, it is ensured without oscillation, we may observe that from Figure 13. The second approach is through the choice of mesh size. From Figure 12 (b), we can see that under difference mesh size, the overshoot behaves differently. Depending on different aspect of oscillation, we may suppress it through different way. From the aspect of peak overshoot, the oscillation may be suppressed through grid with larger mesh size; from the aspect of settling range, the oscillation may be suppressed through grid with smaller mesh size.

4 Conclusion

The conclusion of this task is as followed: First, with high Peclet number, convection dominates diffusion. Second, with respect to first order schemes, second order schemes are shown to have a higher consistency, but a lower stability. Third, smaller mesh-size are shown to have a higher consistency, but not unlimitedly. Furthermore, for high order scheme, oscillation can be suppressed through TVD or the choice of mesh size. Last, stability may be examined through TVD, for TVD schemes its stability is ensured.

5 List of programs

```
<header.h>
//Problem Description
#define PI 3.1415926
#define N 161
#define Pe numeric_limits<double>::infinity()
#define scheme 3 //0:Upwind, 1:2nd Up, 2: CDS, 3: QUICK, 4: minmod
#define SOR 1
#define BC_exchange_rate 1
#define tol pow(10, -9)


void write(double* a, int x, int y);
void printinfo();
void phi_generate(int);

<Source.cpp>
// CDS currently fails at pe>1
#include "header.h"

//Parameters
double m_grid[N][N] = { 0 };
double h = 1 / ((double)N - 1);
double u = 1, v = 1, rho = 1;
double gamma = (double)u / (Pe * (N - 1));
double m_rate = rho * u * h;
double D_e = gamma, D_w = gamma, D_n = gamma, D_s = gamma;
double r_e = 0, r_w = 0, r_n = 0, r_s = 0;
double phi_e = 0, phi_w = 0, phi_n = 0, phi_s = 0;
double A_e, A_w, A_n, A_s, A_ww, A_ss, A_p;
double time2;
int i = 0, j = 0;
int iterations = 0;
double residual = 1;
double residual_after = 1;
double temp = 0;
int partition[10];


int main(int argc, char* argv[]) {
int rank;
int size;
MPI_Status istat[8];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

/*Implementation*/
//CPU distribution
for (i = 0; i <= size; i++) {
partition[i] = (N - 1) / size * i;
}

//Initialization
for (i = 0; i < N; i++) {
m_grid[i][0] = 1.0;
```

```
m_grid[i][N - 1] = 0.0;
m_grid[N - 1][i] = 0.0;
m_grid[0][i] = 1.0;
}

//main
double time1 = MPI_Wtime();
double temp2;
while (residual > tol) {
residual = 0;
iterations++;

/*2nd outer layer by upwind*/
phi_generate(0);
for (i = partition[rank]; i < partition[rank + 1]; i++) {
if (i == N - 2) {
for (j = 1; j < N - 1; j++) {
temp = m_grid[i][j];
m_grid[i][j] = (m_grid[i][j + 1] * A_e + m_grid[i][j - 1] * A_w + m_grid[i - 1][j] * A_n + m_grid[i + 1][j] * A_s) / A_p;
residual = residual + abs(m_grid[i][j] - temp);
}
}

if (i != 0) {
j = 1;
temp = m_grid[i][j];
m_grid[i][j] = (m_grid[i][j + 1] * A_e + m_grid[i][j - 1] * A_w + m_grid[i - 1][j] * A_n + m_grid[i + 1][j] * A_s) / A_p;
residual = residual + abs(m_grid[i][j] - temp);
}
}

//inner
for (i = partition[rank]; i < partition[rank + 1]; i++) {
if ((i == N - 2))
break;

for (j = 2; j < N - 1; j++) {
if (i == 0)
break;

/*Calculate r*/
if (iterations < 10)
phi_generate(0);
else
phi_generate(scheme);
//Calculate numerical solution for the following iteration
temp = m_grid[i][j];
m_grid[i][j] = (m_grid[i][j + 1] * A_e + m_grid[i][j - 1] * A_w + m_grid[i][j - 2] * A_ww + m_grid[i - 1][j] * A_n
+ m_grid[i + 1][j] * A_s + m_grid[i + 2][j] * A_ss) / A_p;
temp2 = m_grid[i][j];
m_grid[i][j] = temp + SOR * (m_grid[i][j] - temp);
residual = residual + abs(m_grid[i][j] - temp);
}
}

MPI_Allreduce(&residual, &residual_after, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
residual = residual_after;

if (iterations % BC_exchange_rate == 0) {
for (i = 0; i < size; i++) {
MPI_Bcast(&m_grid[partition[i]][0], (partition[rank + 1] - partition[rank]) * N, MPI_DOUBLE, i, MPI_COMM_WORLD);
}
}
}
time2 = MPI_Wtime() - time1;

if (rank == 0)
printinfo();
/*-------------*/
MPI_Finalize();
return 0;
}

void printinfo() {
write(&m_grid[0][0], N, N);
cout << "-----------" << endl;
cout << "Peclet number: " << Pe << endl;
cout << "Mesh number:" << N << " Scheme:" << scheme << endl;
cout << "SOR: " << SOR << " BC_exchange_rate:" << BC_exchange_rate << endl;
cout << "Iterations:" << iterations << " residual:" << residual << endl;
cout << "Time:" << time2 << endl;
cout << "-----------" << endl;
return;
}
```

```
void phi_generate(int scheme_implement) {
/*Calculate r*/
r_e = (m_grid[i][j + 1] - m_grid[i][j]) / (m_grid[i][j] - m_grid[i][j - 1]);
r_n = (m_grid[i - 1][j] - m_grid[i][j]) / (m_grid[i][j] - m_grid[i + 1][j]);
r_s = (m_grid[i][j] - m_grid[i + 1][j]) / (m_grid[i + 1][j] - m_grid[i + 2][j]);
r_w = (m_grid[i][j] - m_grid[i][j - 1]) / (m_grid[i][j - 1] - m_grid[i][j - 2]);

/*Calculate phi*/
switch (scheme_implement) {
case 0:
phi_e = 0, phi_n = 0, phi_s = 0, phi_w = 0;
break;
case 1:
phi_e = 1, phi_n = 1, phi_s = 1, phi_w = 1;
break;
case 2:
phi_e = r_e, phi_n = r_n, phi_s = r_s, phi_w = r_w;
break;
case 3:
phi_e = max(0.0, min(2 * r_e, min(2.0, 2 * (r_e + abs(r_e)) / (r_e + 3))));
phi_n = max(0.0, min(2 * r_n, min(2.0, 2 * (r_n + abs(r_n)) / (r_n + 3))));
phi_s = max(0.0, min(2 * r_s, min(2.0, 2 * (r_s + abs(r_s)) / (r_s + 3))));
phi_w = max(0.0, min(2 * r_w, min(2.0, 2 * (r_w + abs(r_w)) / (r_w + 3))));
break;
case 4:
phi_e = max(0.0, min(r_e, 1.0));
phi_n = max(0.0, min(r_n, 1.0));
phi_s = max(0.0, min(r_s, 1.0));
phi_w = max(0.0, min(r_w, 1.0));
break;

}

A_e = D_e;
A_w = ((m_rate * phi_e + m_rate * phi_w) * 0.5 + m_rate) + D_w;
A_n = D_n;
A_s = ((m_rate * phi_n + m_rate * phi_s) * 0.5 + m_rate) + D_s;
A_ww = -m_rate * phi_w * 0.5;
A_ss = -m_rate * phi_s * 0.5;
A_p = A_e + A_w + A_n + A_s + A_ww + A_ss;
return;
}

void write(double *a, int x, int y) {
ofstream myfile("result.csv");
int i,j;
for (i = 0; i < y; i++) {
for (j = 0; j < x; j++) {
myfile << *(a+i*x+j) << ",";
}
myfile << endl;
}

myfile.close();
return;
}
```

6 Reference

Dobrica, M. B., Fillon, M. (2013). Finite Volume Method for Fluid Film Bearings. Encyclopedia of Tribology, 1157–1164.