UNIVERSITY OF OSLO, DEPARTMENT OF MATHEMATICS

Mandatory assignment 1: MEK4470

Trym Erik Nielsen

March 23, 2018

EXCERCISE 1

TVD schemes stand for Total Variation diminishing schemes. TVD attempts to reduce "wiggles" in the numerical solution, by limiting numerical diffusion in the solution. In this exercise, we have attempted to introduce a Van-Leer flux limiter TVD scheme using Central Differencing for the convection, in the one dimensional convection diffusion problem. Below follows the computed results for the transport of ϕ (top graph), and the L_2 computed norm (bottom graph)

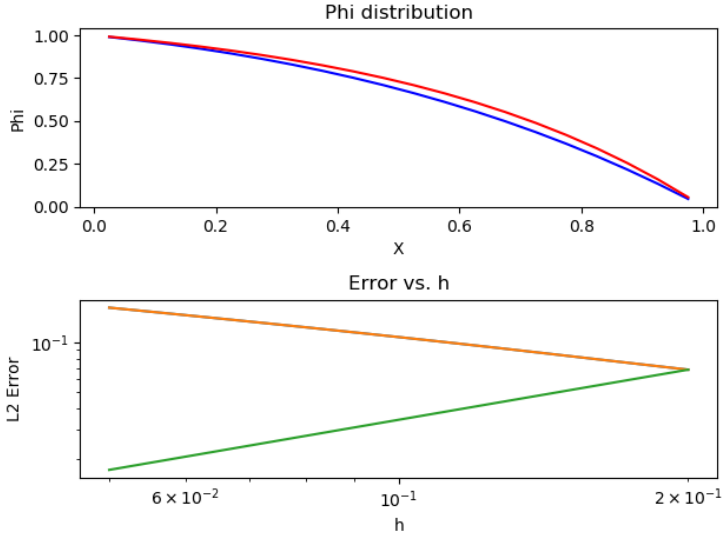

Figure 0.1: Top: Steady state solution ϕ, bottom: L_2 error

We can see from the top graph of figure **??** that we have a reasonable fit with the analytical solution (red) and the computer solution (in blue). However, we do not see convergence in L_2. The log-log graph of the L_2 error versus the cell size shows in an inverse relationship. For convergence of the solution in L_2 we expect L_2 to linearly decrease (on a log-log plot) against smaller cell size. The suspected reason for this is due to the authors difficulty in modelling the ratio of the upstream gradient with the downstream gradient, as well as accounting for all fluxes accross the control volume faces. Refer to appendix for the python code used in this example.

EXCERCISE 2

PART 1

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "pisoControl.H"
#include "fvOptions.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"

    turbulence->validate();

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        #include "CourantNo.H"

        // Pressure-velocity PISO corrector
        {
            #include "UEqn.H"

            // --- PISO loop
            while (piso.correct())
            {
                #include "pEqn.H"
            }
        }

        laminarTransport.correct();
        turbulence->correct();

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}

// ************************************************************************* //
```

Listing 1: Source code for pisoFoam

The Piso algorithm follows the following structure

- Solve the discretized momentum equation
- Solve pressure correction equation
- Correction of pressure and velocities
- solve for second pressure correction
- Solve other transport equations (i.e velocity divergence)
- check for convergence (goto start, if no)
- end

We see from the listing above pisoFoam.C. Within the main time loop is the piso correction code from the header files "UEqn.h" and "pEqn.h". In the header file for the pressure correction we see the equation for the pressure correction

```
fvm::laplacian(rAU, p) == fvc::div(phiHbyA)

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));

    fvOptions.correct(U);
}
```

The PISO algorithm works by maintaining divergence free flow by updating the velocity based on the guess for pressure using the equation mentioned in the previous code snippet. The second corrector step maintains stability of the solution.

PART 2

The equations that model LES in einstein notation:

$$\frac{\partial \bar{u}_i}{\partial t} + u_j \frac{\partial \bar{u}_i}{\partial x_j} = \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{1}{\rho}\frac{\partial \bar{P}}{\partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j}$$

where the term τ_{ij} is used as

$$\tau_{ij} = \overline{u_i u_j} - \bar{u}_i \bar{u}_j$$

for the numerical discretized convection, we have used Gauss LUST, as can be seen in the following snippet of the fvSchemes dictionary:

```
divSchemes
{
    default           none;
    div(phi,U)        Gauss LUST grad(U);
    div(phi,k)        Gauss limitedLinear 1;
    div(phi,s)        bounded Gauss limitedLinear 1;
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}
```

Listing 2: divergence schemes

Gauss LUST mixes Gauss linear and Gauss linearUpwind. This blending is optimized for a range of mesh sizes, and therefore allows us courser mesh, to study the effect of refinement. Upwinding is in general quite stable, and since the flow is quite fast, it is a good scheme for convection.
The boundary conditions used in solving the above using pisoFoam can be found in the github link in the Appendix. In the figures below, the mean values for P and u have been included, both for a corse and a finer mesh created using the openFoam refineMesh utility.

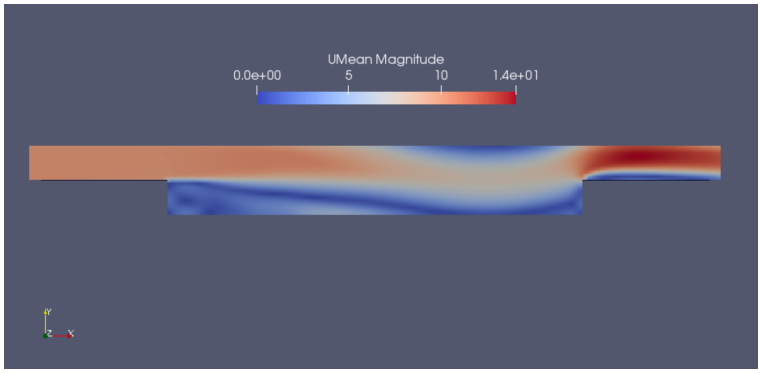

Figure 0.2: Mean velocity magnitude, piso solver

From figure **??** and **??** we can see that even with a doubling of control volume cells, that the mean velocity profile seems quite stable and mesh independent. similarly in figures **??** and figure **??** we see that the mean pressure of the flow seems to also be mesh independent.

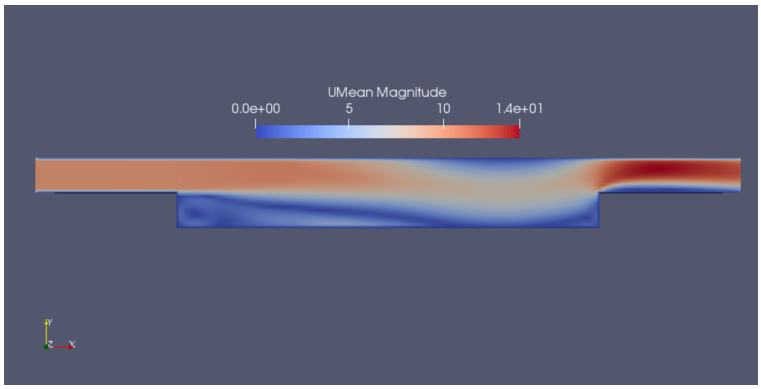

Figure 0.3: Mean velocity magnitude, refined mesh

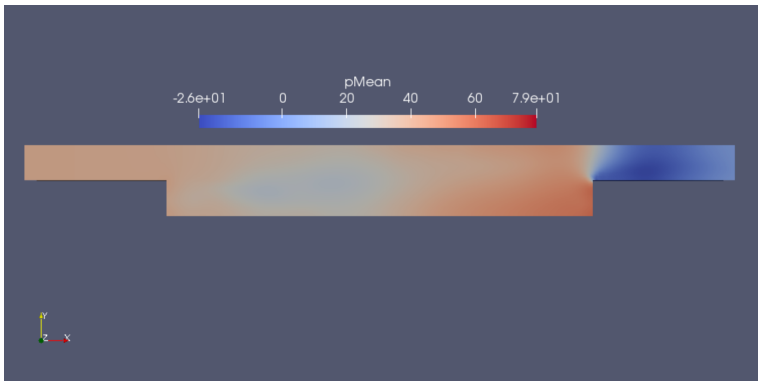

Figure 0.4: Mean Pressure

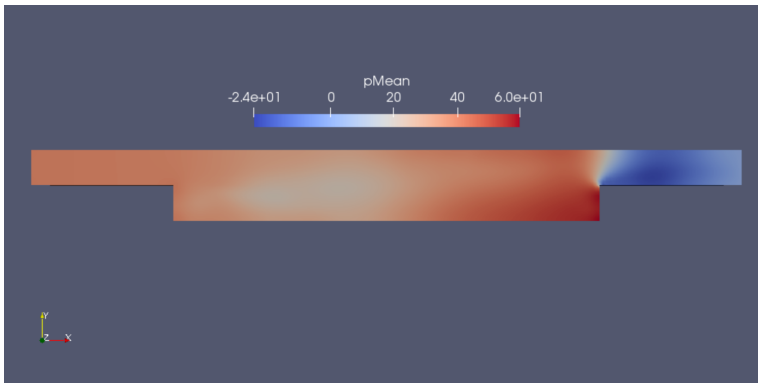

Figure 0.5: Mean Pressure, refined mesh

PART 3

Having found a solution for the flow regime using LES and pisoFoam. We now attempt doing the same using two different RANS models and the simpleFoam solver.
The algorithm of simpleFoam can be found under incompressible solver, as the file simpleFoam.C. The base algorithm without turbulence modelling depends on the functions included from header files in this file, and in perticular the functions from the header files UEqn.H and pEqn.H. Below follows the simpleFoam file.

```
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "simpleControl.H"
#include "fvOptions.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"

    turbulence->validate();

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nStarting time loop\n" << endl;

    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }

        laminarTransport.correct();
        turbulence->correct();

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


// ************************************************************************* //
```

Listing 3: source code for SimpleFoam

We can see in the simple loop structure that the velocity and pressure update and correction functions are included from their respectice header files. The Pressure correction equation can be found in the following code snippet.

```
fvm::laplacian(rAtU(), p) == fvc::div(phiHbyA)
```

and its associated momentum correction equation:

```
U = HbyA - rAtU()*fvc::grad(p);
```

and similarly the function for the velocity guess equation is found in the UEqn.h header file

```
tmp<fvVectorMatrix> tUEqn
(
    fvm::div(phi, U)
  + MRF.DDt(U)
  + turbulence->divDevReff(U)
 ==
    fvOptions(U)
);
```

And velocity is updated in the simpleFoam.C loop, using the equation

```
solve(UEqn == -fvc::grad(p));
```

The boundary conditions used for the simpleFoam run, can be found on the github link in the appendix. In the convection scheme for simpleFoam, we used the default scheme of a bounded linearUpwind, as can be seen in the following snippet of the fvSchemes dictionary:

```
divSchemes
{
    default             none;
    div(phi,U)          bounded Gauss linearUpwind grad(U);
    div(phi,k)          bounded Gauss limitedLinear 1;
    div(phi,epsilon)    bounded Gauss limitedlinear 1;
    div(phi,omega)      bounded Gauss limitedlinear 1;
    div(phi,v2)         bounded Gauss limitedlinear 1;
    div((nuEff*dev2(T(grad(U))))    Gauss linear;
    div(nonlinearStress)    Gauss linear;
}
```

The Gauss linearUpwind scheme is only first order accurate, yet provides high stability. We can choose to run the simulation for as long as required to achieve steady-state, with both a corse and fine mesh.
In the following figures we see the results for mean velocity and mean kinetic energy for both a corse and a finer mesh computed using the $k-\epsilon$. As well as the mean velocity and mean kinetic energy for the $K-\omega$ turbulence model.

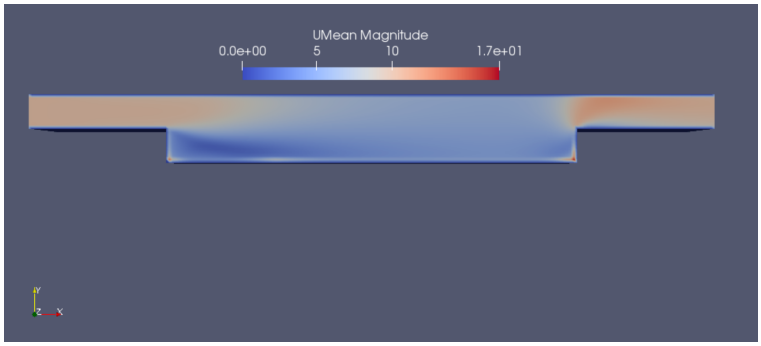

Figure 0.6: Mean velocity magnitude $K-\epsilon$ turbulence

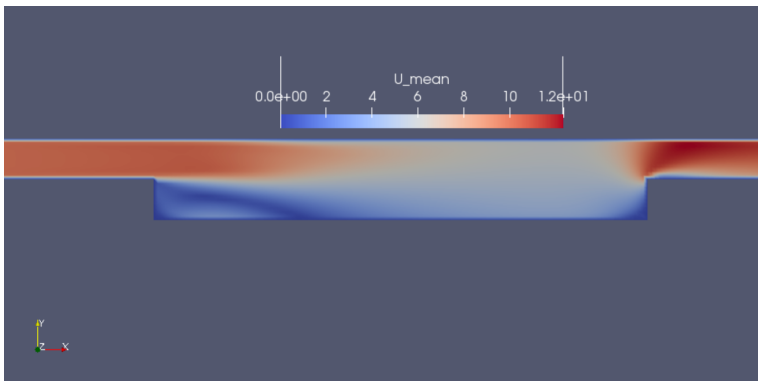

Figure 0.7: Mean velocity magnitude, refined mesh

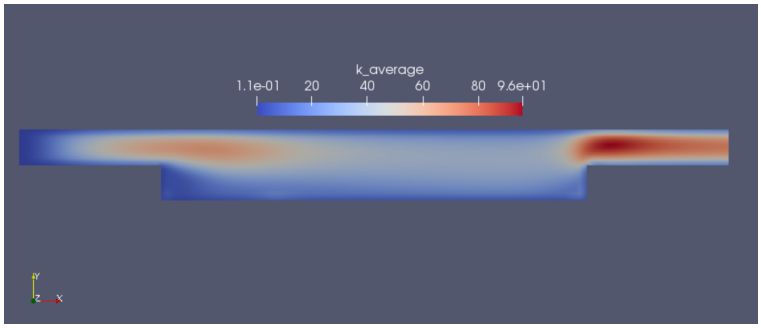

Figure 0.8: Mean kinetic energy $K-\epsilon$ turbulence

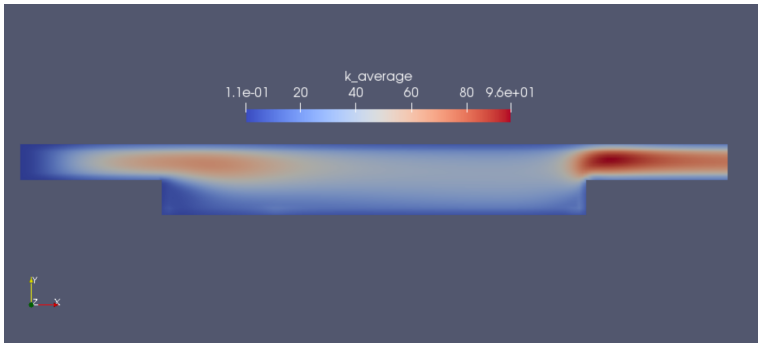

Figure 0.9: Mean kinetic energy, refined mesh

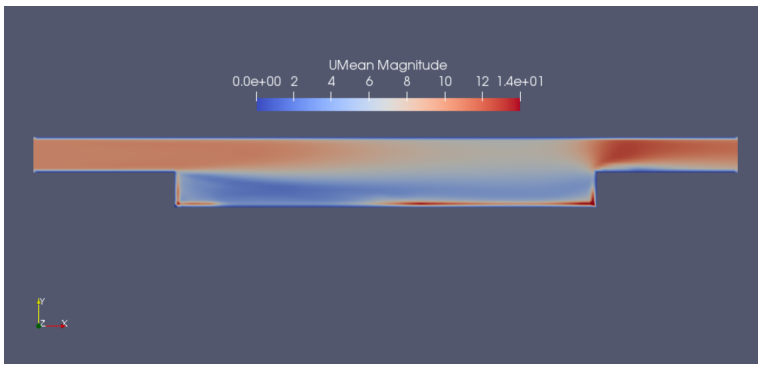

Figure 0.10: Mean velocity magnitude, $K-\omega$ turbulence

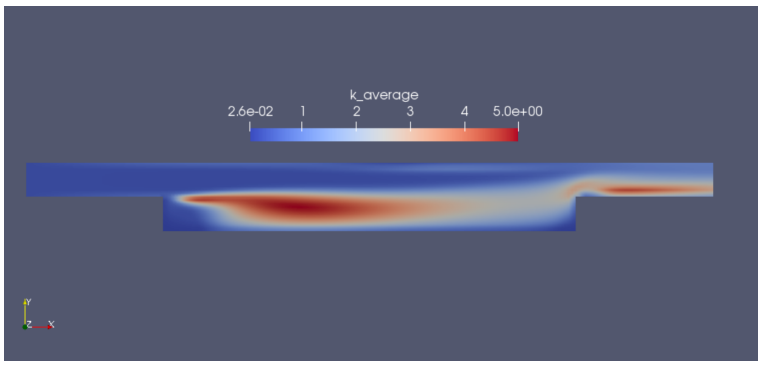

Figure 0.11: Mean kinetic energy, $K-\omega$ turbulence

We can see there is good agreement between the mesh resolutions for both the mean velocity and mean kinetic energy. Interestingly, the $K-\omega$ model shows a different location for the maximum mean kinetic energy. Additionally, we can see some unphysical high velocity flow along the boundaries of the lower left and lower right wall in the $k-\omega$ model. This might be due to an incorrect choice of wallfunction in the boundary conditions.

PART 4

To accurately compute turbulent flow using the LES model, a small Δt is needed, pimpleFoam is however less stable for very small Δt, and is therefore more suitable for RANS modelling.

PART 5

We can get a visual sense of the accuracy of the computed solutions for $\vec{u}$ by comparing the respective locations for the recirculation bubble for both LES and RANS. We can visualize the recirculation bubble by applying contour lines on the velocity magnitude in paraview. Below follows the contour plot for velocity magnitude for LES/pisoFoam.

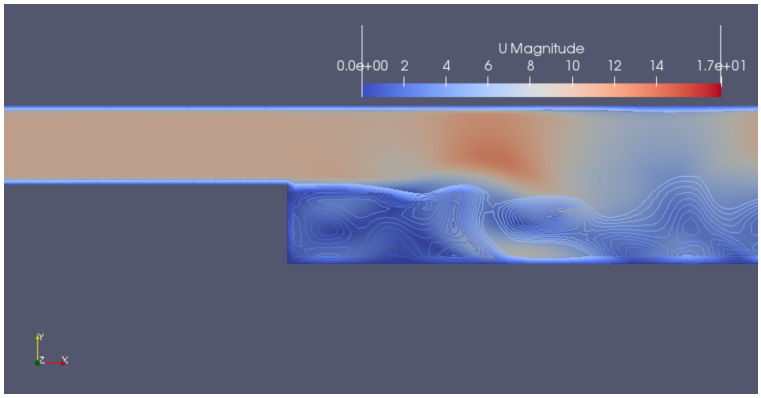

Figure 0.12: Isolines of velocity, pisoFoam

And similarly for the $K-\epsilon$ RANS, SimpleFoam solution:

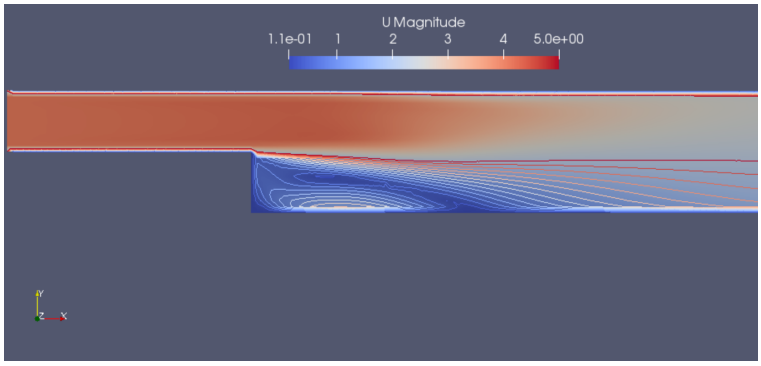

Figure 0.13: Isolines of velocity, simpleFoam

We can see in figure **??** that the RANS solutions is able to accurately model the recirculation bubble as circular. The bubble also appears closer to the left wall, and higher up from the lower wall. In figure **??** we see the bubble has become more smeared out in the direction of flow, and appears closer to the lower wall. This might be because of a difference in how convection is modelled in the respective turbulence models, and also due dissapation of turbulent energy in the case of LES.

PART 6

The method of Large Eddy Simulations is a method of modelling turbulence. By using a corser mesh we can effectively filter out the smaller pertubations in the flow, yet capture the larger turbulent eddies in the flow; hence the name. Turbulence is inherently a three dimensional phenomena, since the LES model computes $\vec{u}(\vec{x},t)$, a 3D geometry is required to accurately model the turbulent flow.

1 APPENDIX

1.1 TVD CODE

```
'''
example 5.1
Upwind/Hybrid/Power-law/QUICK
chapter: 5.6/5.7/5.8/5.9
include L^2 feil
'''
from numpy import *
from pylab import *

def central_diff(N):
    face = linspace(0, 1., N+1)
    #face.shape
    nodes = 0.5*(face[1:] + face[:-1])
    delta = face[1] - face[0]

    # constants
    L = 1.0
    phi_a = 1.0
    phi_b = 0.0
    F = 0.1 # rho*u
    D = 0.1 / (1.0/N) # gamma/dx

    # Set up linear system of equations
    A = zeros((N, N))
    b = zeros(N)

    phi = linspace(1.,0.,N)
    phi[0] = phi_a
    phi[-1] = phi_b

    # ap*T_P - aw*T_W - ae*T_E = 0
    for i in range(1, N-1):
        aw = D + F/2
        ae = D - F/2
        sp = 0
        su = 0
        ap = aw + ae - sp
        A[i, i-1] = -aw
        A[i, i]   = ap
        A[i, i+1] = -ae
        b[i] = su

    # Node 1
    ae = D - F/2
    aw = 0
    sp = -(2*D + F)
    su = (2*D + F) * phi_a
    ap = aw + ae - sp
    A[0, 0] = ap
    A[0, 1] = -ae
    b[0] = su

    # Node N add TVD
    aw = D + F/2
    ae = 0
    sp = -(2*D - F)
    su = (2*D - F) * phi_b
    ap = aw + ae - sp
    A[-1, -1] = ap
    A[-1, -2] = -aw
    b[-1] = su

    return(face,nodes,A,b)
```

```
def Upwind(N):
    faces = linspace(0, 1., N+1)
    nodes = 0.5*(faces[1:] + faces[:-1])
    delta = faces[1]-faces[0]

    # constants
    L = 1.0
    phi_a = 1.0
    phi_b = 0.0
    F = 0.1  # rho*u
    D = 0.1 / (1.0/N) # gamma/dx

    # Set up linear system of equations
    A = zeros((N, N))
    b = zeros(N)

    # ap*T_P - aw*T_W - ae*T_E = 0
    for i in range(1, N-1):
        aw = D + F
        ae = D
        sp = 0
        su = 0
        ap = aw + ae - sp
        A[i, i-1] = -aw
        A[i, i]   = ap
        A[i, i+1] = -ae
        b[i] = su

    # Node 1, TVD corrected
    ae = D
    aw = 0
    sp = -(2*D + F)
    su = (2*D + F) * phi_a
    ap = aw + ae - sp
    A[0, 0] = ap
    A[0, 1] = -ae
    b[0] = su

    # Node N
    aw = D + F
    ae = 0
    sp = -2*D
    su = 2*D*phi_b
    ap = aw + ae - sp
    A[-1, -1] = ap
    A[-1, -2] = -aw
    b[-1] = su
    return nodes,A,b


def QUICK(N):
    faces = linspace(0, 1., N+1)
    nodes = 0.5*(faces[1:] + faces[:-1])
    delta = faces[1]-faces[0]

    # constants
    L = 1.0
    phi_a = 1.0
    phi_b = 0.0
    F = 0.2  # rho*u
    D = 0.1 / (1.0/N) # gamma/dx

    # Set up linear system of equations
    A = zeros((N, N))
    b = zeros(N)
```

```
    for i in range(2, N-1):
        aw = D + (6/8)*F + (1/8)*F
        aww = -(1/8)*F
        ae = D - (3/8)*F
        aee = 0
        ap = aw + ae + aww + aee
        sp = 0
        su = 0
        A[i, i-1] = -aw
        A[i, i-2] = -aww
        A[i, i]   = ap
        A[i, i+1] = -ae
        b[i] = su

    # Node 1
    ae = D + (1/3)*D - (3/8)*F
    aw = 0
    aww = 0
    sp = -((8/3)*D + (2/8)*F + F)
    su = ((8/3)*D + (2/8)*F + F) * phi_a
    ap = aww + aw + ae - sp
    A[0, 0] = ap
    A[0, 1] = -ae
    b[0] = su

    # Node 2
    ae = D - (3/8)*F
    aw = D + (7/8)*F + (1/8)*F
    aww = 0
    sp = 0.25*F
    su = -0.25*F*phi_a
    ap = aww + aw + ae - sp
    A[1, 1] = ap
    A[1, 0] = -aw
    A[1, 2] = -ae
    b[1] = su


    # Node N
    aw = D + (1/3)*D + (6/8)*F
    aww = -(1/8)*F
    ae = 0
    sp = -((8/3)*D - F)
    su = ((8/3)*D - F)*phi_b
    ap = aww + aw + ae - sp
    A[-1, -1] = ap
    A[-1, -2] = -aw
    A[-1, -3] = -aww
    b[-1] = su
    return nodes,A,b

def analytical(x_):
    F=0.2
    return (-1 * (exp(F * (x_/0.1)) - 1) / (exp(F * (1.0/0.1)) - 1)) + 1

def L2_norm(num, anal):
    #return linalg.norm(num - anal)
    return linalg.norm(num-anal,2) #sqrt(1/N * sum((num - anal)**2))

def tvd(N,phi,b,faces):
    #intialize
    F = 0.1 # rho*u
    D = 0.1 / (1.0/N) # gamma/dx
    phi_a = 0
    r_e = 0
    r_w = 0
    lim_e = 0
```

```
    lim_w = 0

    for i in range(1,N-1):
        if i == 1:
            r_w = (2*phi[i] - phi_a) / (phi[i+1] - phi[i])
            lim_w = (r_w + abs(r_w)) / (1 + abs(r_w))
            b[i] += F*(0.5*lim_w*(phi[i] - phi[i-1]))

        r_e = (phi[i] - phi[i-1]) / (phi[i+1] - phi[i]) #Van Leer limiter
        r_w = (phi[i-1] - phi[i-2]) / (phi[i] - phi[i-1])
        lim_e = (r_e + abs(r_e)) / (1 + abs(r_e))
        lim_w = (r_w + abs(r_w)) / (1 + abs(r_w))
        b[i] += -F*(0.5*lim_e*(phi[i+1] - phi[i])) + F*(0.5*lim_w*(phi[i] - phi[i-1]))

    return b

if __name__ == '__main__':
    TVD = True
    err = []
    N = list(range(5,21))

    for i in N:
        face,nodes,A,b = central_diff(i)
        T = solve(A, b)

        if TVD:
            b = tvd(i,T,b,face)
            T = solve(A,b)
            print('TVD solve!')

        exact = analytical(nodes)
        err.append(L2_norm(T,exact))

    figure(1)
    subplot(211)
    plot(nodes, T, 'b')
    plot(nodes, exact, 'r')
    xlabel('X')
    ylabel('Phi')
     title ('Phi distribution')

    subplot(212)
    loglog(1 / asarray(N), err, 'g')
    loglog(1 / asarray(N), err[0]*(1 / asarray(N) * N[0]), 'c')
    #loglog(1 / asarray(N), err[0]*(1 / asarray(N) * N[0]))
    xlabel('h')
    ylabel('L2 Error')
     title ('Error vs. h')
    tight_layout()
    show()
```

Listing 4: Source code for excercise 1

1.2 GITHUB LINK

contains boundary conditions used, source code for Exercise 1, as well as an animation of the flow calculated using pisoFoam

Github.com: Cumulus0