5-MassAndMomentum

March 17, 2022

1 Solution of Mass and Momentum Equations

1.1 Problem Definition

In this lesson, we consider how to solve coupled mass and momentum equations using the finite volume method. There are methods for compressible flows (i.e. density-based methods) that are quite simple to implement, but these don't extend to incompressible flows. We will consider a method for incompressible flows that is extendable to compressible flows (i.e. pressure-based methods)

The equation for conservation of mass, with constant density and no mass sources/sinks is:

$$\nabla \cdot (\rho \mathbf{u}) = 0$$

Conservation of momentum in the x-direction is described by:

$$\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho \mathbf{u} u) = -\frac{dp}{dx} + \nabla \cdot (\mu \nabla u) + f_x$$

where f_x is a body force per unit volume.

1.2 Discretization

Discretization of the conservation of mass equation proceeds as in the previous lesson, resulting in

$$\dot{m}_e - \dot{m}_w = 0$$

Discretization of the momentum equation proceeds by integrating over both space and time, as before.

The transient term is integrated as before, using the space-time method, resulting in

$$\int_{t-\Delta t}^{t+\Delta t} \int_V \frac{\partial (\rho u)}{\partial t} dV dt = (\rho u_P V_P)^{t+\Delta t/2} - (\rho u_P V_P)^{t-\Delta t/2}$$

The interpolation of the time face values at $t - \Delta t/2$ and $t + \Delta t/2$ are conducted as was done previously to derive different schemes (i.e. first order implicit and second order implicit). When the remaining terms in the momentum equation are integrated, they each result in a factor Δt, which ends up being divided through the equation. The discretized trasient term, divided by Δt is then (for constant density and grid topology):

$$\rho V_P \frac{(u_P)^{t+\Delta t/2} - (u_P)^{t-\Delta t/2}}{\Delta t}$$

Time integration of the remaining terms is straightforward (simply resulting in a factor Δt in each term), so only the spatial integration is considered for these terms.

Integration of the advection term is carried out as follows:

$$\int_V \nabla \cdot (\rho \mathbf{u} u)\, dV = \int_S \rho u \mathbf{u} \cdot \mathbf{n} dS \approx \sum_{i=0}^{N_{ip}-1} \rho u \mathbf{u} \cdot \mathbf{n}_{ip} A_{ip}$$

For the one-dimensional grid considered, this becomes:

$$\int_V \nabla \cdot (\rho \mathbf{u} u)\, dV \approx \dot{m}_e u_e - \dot{m}_w u_w$$

The pressure term is integrated by treating it essentially as a source term:

$$-\int_V \frac{dp}{dx} dV \approx -\left.\frac{dp}{dx}\right|_P V_P$$

The viscous term is integrated as:

$$\int_V \nabla \cdot (\mu \nabla u)\, dV = \int_S \mu \nabla u \cdot \mathbf{n} dS$$

The surface integral above represents the viscous forces acting on all faces on a control volume. In the 2D case, this simply becomes a summation over all of the faces; but in the 1D case we separate the normal stresses on the fluid (acting on the east and west faces) and the viscous shear stresses (acting on the north and south faces, which are the boundary of the domain). This results in:

$$\int_V \nabla \cdot (\mu \nabla u)\, dV \approx \sum_{i=0}^{N_{ip}-1} \mu \nabla u \cdot \mathbf{n}_{ip} A_{ip} = \mu \left.\frac{\partial u}{\partial x}\right|_e A_e - \mu \left.\frac{\partial u}{\partial x}\right|_w A_w + \mu \left.\frac{\partial u}{\partial y}\right|_n A_n - \mu \left.\frac{\partial u}{\partial y}\right|_s A_s = \mu \left.\frac{\partial u}{\partial x}\right|_e A_e - \mu \left.\frac{\partial u}{\partial x}\right|_w A_w +$$

where F_u is net viscous shear force acting on the control volume.

Neglecting any other body forces for the time being, the discretized momentum equation is given as:

$$\rho V_P \frac{(u_P)^{t+\Delta t/2} - (u_P)^{t-\Delta t/2}}{\Delta t} + \dot{m}_e u_e - \dot{m}_w u_w = -\left.\frac{dp}{dx}\right|_P V_P + \mu \left.\frac{\partial u}{\partial x}\right|_e A_e - \mu \left.\frac{\partial u}{\partial x}\right|_w A_w + F_u$$

It can be noted that this equation is of the same form as the transport equation previously derived for convection of a scalar, with the addition of the pressure term and the viscous shear term.

The diffusion coefficients are defined similarly to the energy equation, replacing k with μ:

$$D_e = \frac{\mu A_e}{\Delta x_{PE}}$$

$$D_w = \frac{\mu A_w}{\Delta x_{WP}}$$

If we then subtract the conservation of mass equation multiplied by u_P (similar to what was done for the energy equation) and estimate the derivatives in the diffusive terms using a piecewise linear approximation we arrive at:

$$\rho V_P \frac{(u_P)^{t+\Delta t/2} - (u_P)^{t-\Delta t/2}}{\Delta t} + \dot{m}_e(u_e - u_P) - \dot{m}_w(u_w - u_P) = -\left.\frac{dp}{dx}\right|_P V_P + D_e(u_E - u_P) - D_w(u_P - u_W) + F_u$$

If we implement a first-order implicit time integration scheme and the UDS advection scheme, the cell residual is:

$$r_P = \rho V_P \frac{u_P - u_P^o}{\Delta t} + \dot{m}_e \left[\left(\frac{1+\alpha_e}{2}\right) u_P + \left(\frac{1-\alpha_e}{2}\right) u_E - u_P\right] - \dot{m}_w \left[\left(\frac{1+\alpha_w}{2}\right) u_W + \left(\frac{1-\alpha_w}{2}\right) u_P - u_P\right] + \left.\frac{dp}{dx}\right|_P$$

In this case, the linearization coefficients are:

$$a_W = -D_w - \frac{\dot{m}_w}{2}(1+\alpha_w)$$

$$a_E = -D_e + \frac{\dot{m}_e}{2}(1-\alpha_e)$$

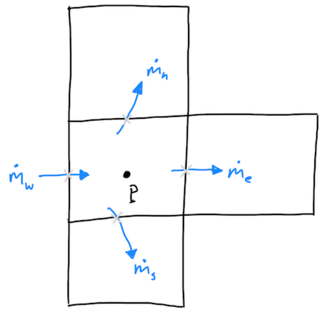
$$a_P = \frac{\rho V_P}{\Delta t} - a_W - a_E$$

Similar to the energy equation, the advection scheme can be improved using deferred corrections, while linearizing based upon UDS for stability.

1.3 The Problem of Pressure-Velocity Coupling

The major issue in solving the mass and momentum equations for an incompressible flow is the fact that pressure does not appear in the conservation of mass equation, since ρ is a constant. Therefore, the idea of pressure-velocity coupling in the context of an incompressible flow, is that the correct pressures are those which drive the velocities (through the momentum equation) to values that conserve mass. Therefore, the conservation of mass equation can be considered as a constraint equation that can be used in determining the correct pressures.

In one dimension, specifying the velocity at the inlet allows us to find the velocity at any other location, provided we know the cross-sectional area at that location. The pressures could then be backed out of the momentum equation using these velocities. In two or more dimensions, the situation is not so simple. If, for example, the mass flux through the west face of a two-dimensional control volume were known, we cannot say how that mass flux will be split among the remaining faces. It is the pressures at the surrounding control volumes that determine how the mass is split. The diagram below illustrates the situation:



Although we are working in one dimension for now, we will not use our knowledge of the cross-sectional area of the domain to determine the mass flows. We will instead develop a method that is valid not only for 1D, but 2D and 3D cases as well.

To illustrate the main issues that occur when coupling the mass and momentum equations, we will consider the simple problem of steady, inviscid flow in a duct with uniform cross-sectional area. The exact solution for this problem is straightforward; both velocity and pressure will be constant.

The discrete mass equation, for a control volume P, for this problem is:

$$\dot{m}_e - \dot{m}_w = 0$$

The discrete momentum equation is:

$$\dot{m}_e u_e - \dot{m}_w u_w = -\left.\frac{dp}{dx}\right|_P V_P$$

Suppose we calculate the mass fluxes using central differences to compute the integration point velocities, i.e.:

$$\dot{m}_e = \rho A_e \left(\frac{u_P + u_E}{2}\right)$$

$$\dot{m}_w = \rho A_w \left(\frac{u_W + u_P}{2}\right)$$

Then, let us compute the pressure gradient at P using the surrounding cell pressures:

$$\left.\frac{dp}{dx}\right|_P = \frac{p_E - p_W}{2\Delta x}$$

where Δx is the grid spacing (i.e. $\Delta x = x_e - x_w$)

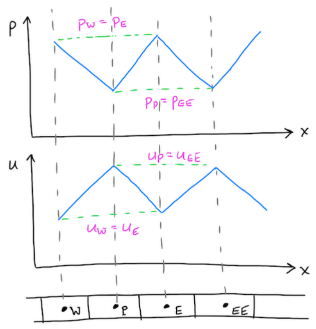
Substituting the above expressions for the mass fluxes into the mass equation, results in:

$$\rho A_e \left(\frac{u_P + u_E}{2}\right) - \rho A_w \left(\frac{u_W + u_P}{2}\right) = 0 \left(\frac{u_P + u_E}{2}\right) - \left(\frac{u_W + u_P}{2}\right) = 0 u_E = u_W$$

Substituting the above expression for the pressure gradient into the momentum equation, and expressing u_e and u_w in terms of the mass fluxes, results in:

$$\frac{\dot{m}_e^2}{\rho A_e} - \frac{\dot{m}_w^2}{\rho A_w} = -\frac{p_E - p_W}{2\Delta x} V_P 0 = -\frac{p_E - p_W}{2\Delta x} V_P p_E = p_W$$

At first glance, this result seems reasonable, since it indicates that the velocity and pressure on either side of the cell P must be equal. The problem, however, is that the velocity and pressure at the cell P are not constrained by these equations. In fact, this solution allows for oscillating pressure and velocity fields in the pattern shown below:



The fact that these oscillatory solutions are accepted by the equation set is a major problem. It means that there is an unconstrained mode in p and u that can grow without bound, while still being accepted as a solution to the problem. That is to say, the solution values at W and E can grow to any value, in relation to the value at P, so long as they keep the same magnitude.

The next question is how to constrain these modes, so that such solutions will not be accepted. One might think that the problem could be due to the CDS approximation that was used in computing the mass fluxes (since this caused so many issues in our advection scheme). To explore that idea, we implement an upwind scheme for evaluation of the mass fluxes (assuming flow in the positive direction):

$$\dot{m}_e = \rho A_e u_P$$

$$\dot{m}_w = \rho A_w u_W$$

The mass equation then results in:

$$\rho A_e u_P - \rho A_w u_W = 0 u_P = u_W$$

This equation will no longer allow the unconstrained velocity mode to exist; only solutions where u is constant everywhere will now be accepted.

In the momentum equation, the left side of the equation still equals zero (since mass is conserved), and results in:

$$p_E = p_W$$

Therefore the same problem still exists for the pressure field. In fact, it can be shown that no matter what approximation method is used for the integration point velocities, the same problem will always exist for the pressure field, if the same approximation is used in the calculation of the mass fluxes (since the left side of the momentum equation will cancel exactly).

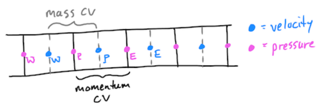
There are two main approaches that can be used to address the problem of pressure decoupling:

- Staggered grids
- Colocated grids

Each of these will be explored subsequently.

1.4 Staggered Grid Methods

The staggered grid method uses two overlapping grids, where one grid stores velocities at its cell centres and the other stores pressures at its cell centres. While the grids overlap, they are shifted by one half of the grid spacing relative to one another, as shown below:



The mass equation is evaluated over the mass control volume (shown in the diagram above). With the staggered grid, the velocities are stored directly at the integration points of the mass control volume, so no interpolation is required. The discrete mass equation for the mass control volume P is then:

$$\dot{m}_e - \dot{m}_w = 0 \rho A_e u_P - \rho A_w u_W = 0 u_W = u_P$$

It should be noted that the subscripts on the velocities are in reference to the momentum control volume labeling. The equation above implies that there can be no decoupling of the velocity field.

The momentum equation is evaluated over the momentum control volume:

$$\dot{m}_e u_e - \dot{m}_w u_w = -\left.\frac{dp}{dx}\right|_P V_P \frac{\dot{m}_e^2}{\rho A_e} - \frac{\dot{m}_w^2}{\rho A_w} = -\frac{p_E - p_P}{\Delta x} V_P 0 = -\frac{p_E - p_P}{\Delta x} V_P p_E = p_P$$

This expression shows that the staggered grid is effecting in removing the possibility of pressure oscillations, preventing decoupling of the velocity and pressure fields.

While staggered grids were popular in the 1980s, their use has nearly disappeared by this point in time. While the one dimensional example shown here is fairly easy to understand and implement, the situation becomes more complex in 2D and 3D cases. The most significant issue, however, is that staggered grids become impractical when considering arbitrary unstructured meshes, which have now become standard for CFD involving more complex geometries.

1.5 Colocated Grid Methods

The work of Rhie and Chow (1983) was instrumental in developing a colocated grid method (i.e. a method where velocity and pressure share the same grid) that was successful in maintaining coupling between velocity and pressure fields. The main idea behind the colocated grid method is that the advected velocity is obtained from a different equation from the advecting velocity. To clarify, the advecting velocity is the one used in the calculation of the mass flux, while the advected velocity is the one that is multiplied by the mass flux in the advection term. The mass flux at the east face is now defined as:

$$\dot{m}_e = \rho A_e \hat{u}_e$$

where $\hat{u}_e$ is the advecting velocity. The advected velocity is simply denoted as u_e as before. The advected velocity is calculated as before, i.e. using the deferred correction approach where the second (or higher) order scheme is linearized with respect to UDS. The advecting velocity $\hat{u}_e$ is now obtained from a special momentum equation that will be derived subsequently. It is desired that $\hat{u}_e \approx u_e$, but that the expressions contain different influences, such that oscillating pressure or velocity fields will be damped out of the solution.

The process of deriving the special momentum equation and the corresponding advecting velocity will be outlined below. The cell residual that was derived previously was:

$$r_P = \rho V_P \frac{(u_P)^{t+\Delta t/2} - (u_P)^{t-\Delta t/2}}{\Delta t} + \dot{m}_e u_e - \dot{m}_w u_w + \left.\frac{dp}{dx}\right|_P V_P - \mu \left.\frac{\partial u}{\partial x}\right|_e A_e + \mu \left.\frac{\partial u}{\partial x}\right|_w A_w - F_u$$

Suppose we have a converged, steady state solution where $r_P = 0$ and the transient term vanishes. The equation then becomes:

$$\dot{m}_e u_e - \dot{m}_w u_w + \left.\frac{dp}{dx}\right|_P V_P - \mu \left.\frac{\partial u}{\partial x}\right|_e A_e + \mu \left.\frac{\partial u}{\partial x}\right|_w A_w - F_u = 0$$

Implementing piecewise profiles for the velocity derivatives, and assuming UDS for the advection terms, results in:

$$\dot{m}_e \left[\left(\frac{1+\alpha_e}{2}\right) u_P + \left(\frac{1-\alpha_e}{2}\right) u_E - u_P\right] - \dot{m}_w \left[\left(\frac{1+\alpha_w}{2}\right) u_W + \left(\frac{1-\alpha_w}{2}\right) u_P - u_P\right] + \left.\frac{dp}{dx}\right|_P V_P + D_w(u_P - u_W) -$$

or:

$$\left[D_w + \frac{\dot{m}_w}{2}(1+\alpha_w) + D_e - \frac{\dot{m}_e}{2}(1-\alpha_e)\right] u_P + \left[-D_w - \frac{\dot{m}_w}{2}(1+\alpha_w)\right] u_W + \left[-D_e + \frac{\dot{m}_e}{2}(1-\alpha_e)\right] u_E - F_u + \left.\frac{dp}{dx}\right|_P$$

Recalling the definitions of the linearization coefficients:

$$a_W = -D_w - \frac{\dot{m}_w}{2}(1+\alpha_w)$$

$$a_E = -D_e + \frac{\dot{m}_e}{2}(1-\alpha_e)$$

$$a_P = \frac{\rho V_P}{\Delta t} - a_W - a_E$$

Then we can define:

$$\overline{a}_P = -a_W - a_E$$

which removes the timestep dependence from the linearization coefficient on P. This is a slight departure from the work of Rhie and Chow in order to make the pressure-velocity coupling independent of the timestep.

Then, the momentum equation can be expressed as:

$$\overline{a}_P u_P + a_W u_W + a_E u_E - b_P + \left.\frac{dp}{dx}\right|_P V_P = 0$$

where b_P can, in general, contain all body force terms present in the equation. Next, let us define:

$$\overline{u}_P = -a_W u_W - a_E u_E + b_P$$

or, more generally:

$$\overline{u}_P = -\sum_{nb} a_{nb} u_{nb} + b_P$$

where *nb* refers to all neighbouring cells (i.e. for 2D or 3D cases). With this notation, the momentum equation becomes:

$$\overline{a}_P u_P = \overline{u}_P - \left.\frac{dp}{dx}\right|_P V_P$$

For the control volume to the east, we can say:

$$\overline{a}_E u_E = \overline{u}_E - \left.\frac{dp}{dx}\right|_E V_E$$

Then, by analogy, for a "virtual" control volume located at the east face integration point we can say:

$$\overline{a}_e \hat{u}_e = \overline{u}_e - \left.\frac{dp}{dx}\right|_e V_e$$

The equation above defines the advecting velocity. The quantity $\overline{u}_e$ is obtained by central differencing from the P and E values, i.e.:

$$\overline{u}_e = \frac{1}{2}\left(\overline{u}_P + \overline{u}_E\right) = \frac{1}{2}\left(\overline{a}_P u_P + \overline{a}_E u_E + \left.\frac{dp}{dx}\right|_P V_P + \left.\frac{dp}{dx}\right|_E V_E\right)$$

To simplify the expression, we make the following approximations within the $\overline{u}_e$ term:

$$\overline{a}_P \approx \overline{a}_E \approx \overline{a}_e$$

$$V_P \approx V_E \approx V_e$$

While these approximations are not strictly necessary, they do substantially simplify the resulting expression. With these approximations:

$$\overline{u}_e = \frac{\overline{a}_e}{2}\left(u_P + u_E\right) + \frac{V_e}{2}\left(\left.\frac{dp}{dx}\right|_P + \left.\frac{dp}{dx}\right|_E\right)$$

The expression for the advecting velocity is then:

$$\overline{a}_e \hat{u}_e = \frac{\overline{a}_e}{2}\left(u_P + u_E\right) + \frac{V_e}{2}\left(\left.\frac{dp}{dx}\right|_P + \left.\frac{dp}{dx}\right|_E\right) - \left.\frac{dp}{dx}\right|_e V_e$$

or:

$$\hat{u}_e = \frac{1}{2}\left(u_P + u_E\right) - \frac{V_e}{\overline{a}_e}\left[\left.\frac{dp}{dx}\right|_e - \frac{1}{2}\left(\left.\frac{dp}{dx}\right|_P + \left.\frac{dp}{dx}\right|_E\right)\right]$$

Let us then define:

$$\overline{a}_e \approx \frac{1}{2}\left(\overline{a}_P + \overline{a}_E\right)$$

$$V_e \approx \frac{1}{2}\left(V_P + V_E\right)$$

$$\hat{d}_e = \frac{V_e}{\overline{a}_e}$$

Note that for a cell with index i, the value of $\overline{a}_E$ is $\overline{a}_P$ at the index $i+1$ (not to be confused with the linearization coefficient a_E). With these definitions, the advecting velocity is defined as:

$$\hat{u}_e = \frac{1}{2}(u_P + u_E) - \hat{d}_e \left[\left.\frac{dp}{dx}\right|_e - \frac{1}{2}\left(\left.\frac{dp}{dx}\right|_P + \left.\frac{dp}{dx}\right|_E \right) \right]$$

The first term on the right side of the equation above is a CDS approximation of the east face velocity. This was the estimate that we had previously used to calculate the mass flux which lead to the emergence of the unconstrained velocity and pressure modes. The second term on the right is a 4th order pressure correction term that serves to smooth out any oscillations that start to emerge in the pressure field. It is called a 4th order correction because the evaluation of the pressure gradient terms will involve the cell values of pressure at the locations W, P, E, and EE. The coefficient $\hat{d}_e$ can be interpreted as a sort of relaxation parameter. In general, the pressure term should be small, such that the advecting velocity is close to the CDS interpolation based on the adjacent cell values. If, however, a pressure oscillation begins to emerge, the pressure terms will be activated and will cause it to be damped out.

The pressure gradient at the east face is calculated as

$$\left.\frac{dp}{dx}\right|_e = \frac{p_E - p_P}{\Delta x_{PE}}$$

while the cell pressure gradients are calculated as:

$$\left.\frac{dp}{dx}\right|_P = \frac{p_E - p_W}{\Delta x_{WP} + \Delta x_{PE}}$$

$$\left.\frac{dp}{dx}\right|_E = \frac{p_{EE} - p_P}{\Delta x_{PE} + \Delta x_{E,EE}}$$

1.6 Coupled (Direct) Solution Method

The discretized momentum equation has been presented already. The only modification required is to compute the mass fluxes using the advecting velocity as it was derived above. The discretized conservation of mass equation can be given in terms of the advecting velocity as:

$$\rho A_e \hat{u}_e - \rho A_w \hat{u}_w = 0 \rho A_e \frac{1}{2}(u_P + u_E) - \rho A_e \hat{d}_e \left[\frac{p_E - p_P}{\Delta x_{PE}} - \frac{1}{2}\left(\left.\frac{dp}{dx}\right|_P + \left.\frac{dp}{dx}\right|_E \right) \right] - \rho A_w \frac{1}{2}(u_W + u_P) + \rho A_w \hat{d}_w \left[\frac{p_P - p_W}{\Delta x_{WP}} \right.$$

Collecting all of the pressure and velocity terms together, and expressing as a residual for the mass equation:

$$r_P = \left[\frac{\rho A_e \hat{d}_e}{\Delta x_{PE}} + \frac{\rho A_w \hat{d}_w}{\Delta x_{WP}} \right] p_P + \left[\frac{\rho A_e}{2} - \frac{\rho A_w}{2} \right] u_P - \left[\frac{\rho A_w \hat{d}_w}{\Delta x_{WP}} \right] p_W - \left[\frac{\rho A_w}{2} \right] u_W - \left[\frac{\rho A_e \hat{d}_e}{\Delta x_{PE}} \right] p_E + \left[\frac{\rho A_e}{2} \right] u_E - \frac{\rho A_w \hat{d}_w}{2} \left[\right.$$

It can now be seen that the pressure has been introduced into the mass equation, which allows us to solve for both pressure and velocity simultaneously, since we have two equations with two unknowns. We shall linearize with respect to both pressure and velocity, but leave the final two terms as explicit "lagged" correction terms to eliminate decoupling.

Let the solution variable be the vector $[p, u]$, such that the linearized problem is given as:

$$\begin{bmatrix} a_P^{pp} & a_P^{pu} \\ a_P^{up} & a_P^{uu} \end{bmatrix} \begin{Bmatrix} \delta p_P \\ \delta u_P \end{Bmatrix} + \begin{bmatrix} a_W^{pp} & a_W^{pu} \\ a_W^{up} & a_W^{uu} \end{bmatrix} \begin{Bmatrix} \delta p_W \\ \delta u_W \end{Bmatrix} + \begin{bmatrix} a_E^{pp} & a_E^{pu} \\ a_E^{up} & a_E^{uu} \end{bmatrix} \begin{Bmatrix} \delta p_E \\ \delta u_E \end{Bmatrix} = - \begin{Bmatrix} r_P^p \\ r_P^u \end{Bmatrix}$$

The superscripts on the linearization coefficients denote the equation and variable to which the coefficient is associated. The first row in the matrix system represents and mass equation (denoted by p) and the second row represents the momentum equation (denoted by u). The first letter in the superscript represents the row (i.e. the equation) and the second represents the column (i.e. the variable). For example, the superscript pu represents the veriable of velocity in the mass equation.

The linearization coefficients for velocity in the momentum equation have already been established:

$$a_W^{uu} = -D_w - \frac{\dot{m}_w}{2}(1 + \alpha_w)$$

$$a_E^{uu} = -D_e + \frac{\dot{m}_e}{2}(1 - \alpha_e)$$

$$a_P^{uu} = \frac{\rho V_P}{\Delta t} - a_W^{uu} - a_E^{uu}$$

The pressure gradient term in the momentum equation can be discretized as:

$$\left.\frac{dp}{dx}\right|_P V_P = \frac{1}{2}\left[\frac{p_P - p_W}{\Delta x_{WP}} + \frac{p_E - p_P}{\Delta x_{PE}}\right] V_P$$

The linearization coefficients for pressure in the momentum equation are therefore:

$$a_W^{up} = -\frac{V_P}{2\Delta x_{WP}}$$

$$a_E^{up} = \frac{V_P}{2\Delta x_{PE}}$$

$$a_P^{up} = -a_W^{up} - a_E^{up}$$

This completes the linearization of the momentum equation.

Linearization of the mass equation is based up on the residual form of the mass equation given above. For linearization with respect to pressure this results in:

$$a_W^{pp} = -\frac{\rho A_w \hat{d}_w}{\Delta x_{WP}}$$

$$a_E^{pp} = -\frac{\rho A_e \hat{d}_e}{\Delta x_{PE}}$$

$$a_P^{pp} = -a_W^{pp} - a_E^{pp}$$

For linearization with respect to velocity, it results in:

$$a_W^{pu} = -\frac{\rho A_w}{2}$$

$$a_E^{pu} = \frac{\rho A_e}{2}$$

$$a_P^{pu} = a_W^{pu} + a_E^{pu}$$

With this, the linearization of the matrix system is complete, since each element of the coefficient matrices has now been filled. It should be noted, however, that these equations are only valid in control volumes where the advecting velocity is computed based on the special momentum equation (i.e. interior faces only). At the boundaries, the special momentum equation is not needed, so the mass equation needs to be modified.

Consider the left boundary control volume, where the velocity $u_w = u_W$ is specified through a boundary condition. The mass equation is then:

$$\rho A_e \hat{u}_e - \rho A_w u_W = 0$$

where the special momentum equation is used only in the calculation of the east face advecting velocty. Carrying through the procedure in a similar manner as before, results in:

$$\rho A_e \frac{1}{2}\left(u_P + u_E\right) - \rho A_e \hat{d}_e \left[\frac{p_E - p_P}{\Delta x_{PE}} - \frac{1}{2}\left(\left.\frac{dp}{dx}\right|_P + \left.\frac{dp}{dx}\right|_E\right)\right] - \rho A_w u_W = 0$$

$$a_W^{pp} = 0$$

$$a_E^{pp} = -\frac{\rho A_e \hat{d}_e}{\Delta x_{PE}}$$

$$a_P^{pp} = -a_E^{pp}$$

$$a_W^{pu} = -\rho A_w$$

$$a_E^{pu} = \frac{\rho A_e}{2}$$

$$a_P^{pu} = a_E^{pu}$$

Exercise: Derive the linearization coefficients for the control volume located at the east boundary.

1.7 Segregated Solution Method

The segregated solution method solves the mass and momentum equations separately, i.e. not in the same matrix. This method can be easier to code, and may provide better convergence in cases where the matrix resulting from the coupled is very stiff (i.e. difficult to solve numerically). The basic process for the segregated solution method is summarized below.

1.7.1 Step 1 - Momentum Solution

For the best available estimate of the pressure, call this p^*, we solve the momentum equation for u^*. We give these variables special symbols, rather than calling them p and u because they will not generally conserve mass until the system is iterated several times. We can define the actual velocity field, u, through the equation:

$$u = u^* + u'$$

where u' is the correction required to get the initial velocity solution, u^*, to conserve mass. The discrete momentum equation can be written, as before, as:

$$a_P u_P^* = \sum a_{nb} u_{nb}^* + b_P - \frac{p_E^* - p_W^*}{x_E - x_W} V_P$$

This equation is solved by linearizing and forming the appropriate linear system, as we have done before. Unless $p^* = p$ (i.e. it is the correct pressure field), then the u^* fields will not be correct because mass will not be conserved.

1.7.2 Step 2 - Evaluate Mass Conservation

Since the field u^* may not converve mass (until convergence), the next step is to evaluate the mass imbalance by calculating the advecting velocities. The calculation is similar to what was derived previously, except that it uses the approximate velocities and pressure (u^* and p^*) in place of u and p:

$$\hat{u}_e^* = \frac{1}{2}\left(u_P^* + u_E^*\right) - \hat{d}_e \left[\left.\frac{dp^*}{dx}\right|_e - \frac{1}{2}\left(\left.\frac{dp^*}{dx}\right|_P + \left.\frac{dp^*}{dx}\right|_E \right) \right]$$

Using the values of $\hat{u}^*$ we can check if mass is conserved, i.e. if $\rho A_e \hat{u}_e^* - \rho A_w \hat{u}_w^* \approx 0$. If this is satisfied to within a suitable tolerance, then the solution is complete. If it is not satisfied, then the pressure needs to be corrected such that it drives the velocity to a state that conserves mass.

1.7.3 Step 3 - Calculate Pressure Correction

Similar to the velocity field, we can define the actual pressure field as the sum of the approximate pressure field, plus a correction:

$$p = p^* + p'$$

As such, we need to derive an equation for p'. To do so, we start with convervation of mass:

$$\rho A_e \hat{u}_e - \rho A_w \hat{u}_w = 0$$

where:

$$\hat{u} = \hat{u}^* + \hat{u}'$$

Substituting this into the conservation of mass equation:

$$\rho A_e \hat{u}_e^* - \rho A_w \hat{u}_w^* = -\left(\rho A_e \hat{u}_e' - \rho A_w \hat{u}_w'\right)$$

The goal is to determine how to correct p, given the required velocity corrections, $\hat{u}'$, in order to satisfy the mass equaiton. Consider the momentum equations for $\hat{u}$ and $\hat{u}^*$, written by analogy for the east face:

$$a_e \hat{u}_e = \left(\sum a_{nb} u_{nb}\right)_e + b_e - \frac{p_E - p_P}{\Delta x_{PE}} V_e$$

$$a_e \hat{u}_e^* = \left(\sum a_{nb} u_{nb}^*\right)_e + b_e - \frac{p_E^* - p_P^*}{\Delta x_{PE}} V_e$$

Subtracting the second equation above from the first results in:

$$a_e \hat{u}_e' = \left(\sum a_{nb} u_{nb}'\right)_e - \frac{p_E' - p_P'}{\Delta x_{PE}} V_e$$

The equation above is used to obtain a connection between u' and p', but additional assumptions need to be made to simplify the expression to make it useful.

The SIMPLE (Semi-Implicit Method for Pressure Linked Equations) method of Spalding and Patankar assumes that $u_{nb}' = 0$, which results in:

$$\hat{u}_e' = -\frac{p_E' - p_P'}{\Delta x_{PE}} \frac{V_e}{a_e}$$

The SIMPLEC (SIMPLE-Consistent) method of Van Doormal and Raithby assumes that $u_{nb}' = \hat{u}_e'$, which results in:

$$\hat{u}'_e = -\frac{p'_E - p'_P}{\Delta x_{PE}} \frac{V_e}{a_e - \sum a_{nb}}$$

To simplify the notation, let us define:

$$\hat{u}'_e = -\left(p'_E - p'_P\right) \hat{c}_e$$

where:

$$\hat{c}_e = \begin{cases} \frac{V_e/\Delta x_{PE}}{a_e} & \text{for SIMPLE} \\ \frac{V_e/\Delta x_{PE}}{a_e - \sum a_{nb}} & \text{for SIMPLEC} \end{cases}$$

These two methods essentially represent two extremes in appromximation, where the actual condition lies somewhere in between. It can be shown that SIMPLE gives an upper bound on the p' values and SIMPLEC gives a lower bound. The fact that SIMPLE is an upper bound explains why this scheme commonly requires relaxation in practice to avoid changing the pressure too rapidly. SIMPLEC can gives close to optimal convergence and is recommended in general in favour of SIMPLE.

If we now substitute the $\hat{u}'_e$ expressions into the conservation of mass equation, we obtain an expression for p':

$$\rho A_e \hat{u}^*_e - \rho A_w \hat{u}^*_w = \rho A_e \left(p'_E - p'_P\right) \hat{c}_e - \rho A_w \left(p'_P - p'_W\right) \hat{c}_w$$

or

$$\left(\rho A_w \hat{c}_w + \rho A_e \hat{c}_e\right) p'_P - \rho A_w \hat{c}_w p'_W - \rho A_e \hat{c}_e p'_E = -\rho A_e \hat{u}^*_e + \rho A_w \hat{u}^*_w$$

If the right side of the equation is positive (i.e. more mass coming into the control volume than leaving), then p'_P will generally be larger than p'_W and p'_E, meaning that p_P increases relative to the surrounding pressures. This will cause less flow into the control volume and more flow out of the control volume, bringing the system closer to conserving mass.

1.7.4 Step 4 - Correct Pressure and Velocity Fields

The final step in the segregated solution method is to correct the pressure and velocity fields according to:

$$p = p^* + p'$$

and

$$\hat{u}_e = \hat{u}^*_e - \hat{c}_e \left(p'_E - p'_P\right)$$

After this step is complete, the algorithm would return to Step 1. The corrected pressure p would then become the current best estimate $p*$ for the next iteration. The advecting velocities will have also been corrected (ideally so that they better conserve mass) and the next solution will be closer to the correct solution. This process is repeated until mass is suitably conserved.

1.7.5 Summary

Note that the application of Steps 1-4 replaces one call to the coupled solver, when using the coupled solution approach. The linearization loop still needs to be completed, as well as the time loop (if applicable). During the application of the segregated algorithms for pressure-velocity coupling, the linearization coefficients are not updated.

While the linear system is smaller in the segregated system (just one variable per control volume at a time), if needs to be solved repeatedly. Therefore, selection of the appropriate method often comes down to the stiffness of the coupled linear system, and how difficult it is to solve numerically.

1.8 Implementation

We will now implement the code for the coupled pressure-velocity method. This will require a number of new classes to be created, as well as some modifications of some old ones.

First, we will modify the transient model to be more general, by taking a generic variable ϕ and by passing an optional constant. The constant will be equal to c_p when solving the energy equation and equal to 1 when solving momenum. For now we will only consider the first order transient model, since we are focusing mainly on steady-state problems in this lesson (so time accuracy is not important). The modified class is shown below:

[4]:
```python
import numpy

class FirstOrderTransientModel:
    """Class defining a first order implicit transient model"""

    def __init__(self, grid, phi, phiold, rho, const, dt):
        """Constructor"""
        self._grid = grid
        self._phi = phi
        self._phiold = phiold
        self._rho = rho
        self._const = const
        self._dt = dt

    def add(self, coeffs):
        """Function to add transient term to coefficient arrays"""

        # Calculate the transient term
        transient = self._rho*self._const*self._grid.vol*(self._phi[1:-1]-self.
↪_phiold[1:-1])/self._dt

        # Calculate the linearization coefficient
        coeff = self._rho*self._const*self._grid.vol/self._dt

        # Add to coefficient arrays
        coeffs.accumulate_aP(coeff)
        coeffs.accumulate_rP(transient)

        return coeffs
```

We make the same modification to the upwind advection scheme to make it more general, as shown below.

[5]:
```
class UpwindAdvectionModel:
    """Class defining an upwind advection model"""

    def __init__(self, grid, phi, Uhe, rho, const, west_bc, east_bc):
        """Constructor"""
        self._grid = grid
        self._phi = phi
        self._Uhe = Uhe
        self._rho = rho
        self._const = const
        self._west_bc = west_bc
        self._east_bc = east_bc
        self._alphae = np.zeros(self._grid.ncv+1)
        self._phie = np.zeros(self._grid.ncv+1)

    def add(self, coeffs):
        """Function to add diffusion terms to coefficient arrays"""

        # Calculate the weighting factors
        for i in range(self._grid.ncv+1):
            if self._Uhe[i] >= 0:
                self._alphae[i] = 1
            else:
                self._alphae[i] = -1

        # Calculate the east integration point values (including both␣
↪boundaries)
        self._phie = (1 + self._alphae)/2*self._phi[0:-1] + (1 - self._alphae)/
↪2*self._phi[1:]

        # Calculate the face mass fluxes
        mdote = self._rho*self._Uhe*self._grid.Af

        # Calculate the west and east face advection flux terms for each face
        flux_w = self._const*mdote[:-1]*self._phie[:-1]
        flux_e = self._const*mdote[1:]*self._phie[1:]
        # Calculate mass imbalance term
        imbalance = - self._const*mdote[1:]*self._phi[1:-1] + self.
↪_const*mdote[:-1]*self._phi[1:-1]

        # Calculate the linearization coefficients
        coeffW = - self._const*mdote[:-1]*(1 + self._alphae[:-1])/2
        coeffE = self._const*mdote[1:]*(1 - self._alphae[1:])/2
        coeffP = - coeffW - coeffE

        # Modify the linearization coefficients on the boundaries
        coeffP[0] += coeffW[0]*self._west_bc.coeff()
        coeffP[-1] += coeffE[-1]*self._east_bc.coeff()

        # Zero the boundary coefficients that are not used
        coeffW[0] = 0.0
        coeffE[-1] = 0.0

        # Calculate the net flux from each cell
        flux = flux_e - flux_w

        # Add to coefficient arrays
        coeffs.accumulate_aP(coeffP)
        coeffs.accumulate_aW(coeffW)
        coeffs.accumulate_aE(coeffE)
        coeffs.accumulate_rP(flux)
        coeffs.accumulate_rP(imbalance)

        # Return the modified coefficient array
        return coeffs
```

Exercise: Modify your CDS and QUICK advection schemes to take a generic scalar field and a coefficient, similar to the UDS scheme shown above.

Next, we need to develop a model for the advecting velocity, shown below. A brief description of the input variables to the constructor is provided below:

Variable	Description	Array dimension
`grid`	Object defining the grid	N/A
`dhat`	Array of damping coefficients at the east faces	`ncv+1`
`Uhe`	Array of advecting velocities at the east faces	`ncv+1`
`P`	Array of cell pressures	`ncv+2`
`U`	Array of cell velocities	`ncv+2`
`U`	Array of cell velocities	`ncv+2`
`coeffs`	Object defining the momentum linearization coeffs	N/A

All arrays are stored by reference in the class. Therefore we have to make sure that we only modify their values and do not bind them to another object.

The `update` method updates the values of `dhat` and `Uhe`, which will be created in the main body of the code.

[6]:
```python
class AdvectingVelocityModel:
    """Class defining an advecting velocity model"""

    def __init__(self, grid, dhat, Uhe, P, U, coeffs):
        """Constructor"""
        self._grid = grid
        self._dhat = dhat
        self._Uhe = Uhe
        self._P = P
        self._U = U
        self._coeffs = coeffs

    def update(self):
        """Function to update the advecting velocity array"""

        # Calculate the pressure gradients across the faces
        gradPw = (self._P[1:-1]-self._P[0:-2])/self._grid.dx_WP
        gradPe = (self._P[2:]-self._P[1:-1])/self._grid.dx_PE

        # Calculate the cell pressure gradients
        gradP = 0.5*(gradPw + gradPe)

        # Calculate damping coefficient, dhat
        Ve = 0.5*(self._grid.vol[0:-1] + self._grid.vol[1:])
        ae = 0.5*(self._coeffs.aP[0:-1] + self._coeffs.aP[1:])
        self._dhat[1:-1] = Ve/ae

        # Update the advecting velocity
        self._Uhe[0] = self._U[0]
        self._Uhe[1:-1] = 0.5*(self._U[1:-2] + self._U[2:-1]) - self._dhat[1:
↪-1]*(gradPe[:-1] - 0.5*(gradP[:-1] + gradP[1:]))
        self._Uhe[-1] = self._U[-1]
```

Next, we need to develop a model to add the pressure terms to the momentum equation, as shown below. The variables passed to the constructor are summarized as:

Variable	Description	Array dimension
`grid`	Object defining the grid	N/A
`P`	Array of cell pressures	`ncv+2`
`west_bc`	Object defining west face boundary condition	N/A
`east_bc`	Object defining east face boundary condition	N/A

[7]:
```python
class PressureForceModel:
    """Class defining a pressure force model"""

    def __init__(self, grid, P, west_bc, east_bc):
        """Constructor"""
        self._grid = grid
        self._P = P
        self._west_bc = west_bc
        self._east_bc = east_bc

    def add(self, coeffs):
        """Function to add diffusion terms to coefficient arrays"""

        # Calculate the pressure force
        gradPw = (self._P[1:-1]-self._P[0:-2])/self._grid.dx_WP
        gradPe = (self._P[2:]-self._P[1:-1])/self._grid.dx_PE
        force = 0.5*(gradPw + gradPe)*self._grid.vol

        # Calculate the linearization coefficients
        coeffW = - 0.5*self._grid.vol/self._grid.dx_WP
        coeffE = 0.5*self._grid.vol/self._grid.dx_PE
        coeffP = - coeffW - coeffE

        # Modify the linearization coefficients on the boundaries
        coeffP[0] += coeffW[0]*self._west_bc.coeff()
        coeffP[-1] += coeffE[-1]*self._east_bc.coeff()

        # Zero the boundary coefficients that are not used
        coeffW[0] = 0.0
        coeffE[-1] = 0.0

        # Add to coefficient arrays
        coeffs.accumulate_aP(coeffP)
        coeffs.accumulate_aW(coeffW)
        coeffs.accumulate_aE(coeffE)
        coeffs.accumulate_rP(force)

        # Return the modified coefficient array
        return coeffs
```

The mass conservation equation is defined in the class `MassConservationModel` as shown below. The variables required for the constructor are summarized as:

Variable	Description	Array dimension
`grid`	Object defining the grid	N/A
`U`	Array of cell velocities	`ncv+2`
`P`	Array of cell pressures	`ncv+2`

Variable	Description	Array dimension
dhat	Array of damping coefficients at the east faces	ncv+1
Uhe	Array of advecting velocities at the east faces	ncv+1
rho	Density	N/A
P_west_bc	Object defining west face pressure boundary condition	N/A
P_east_bc	Object defining east face pressure boundary condition	N/A
U_west_bc	Object defining west face velocity boundary condition	N/A
U_east_bc	Object defining east face velocity boundary condition	N/A

[8]:
```
class MassConservationEquation:
    """Class defining a mass conservation equation"""

    def __init__(self, grid, U, P, dhat, Uhe, rho,
                 P_west_bc, P_east_bc, U_west_bc, U_east_bc):
        """Constructor"""
        self._grid = grid
        self._U = U
        self._P = P
        self._dhat = dhat
        self._Uhe = Uhe
        self._rho = rho
        self._P_west_bc = P_west_bc
        self._P_east_bc = P_east_bc
        self._U_west_bc = U_west_bc
        self._U_east_bc = U_east_bc

    def add(self, PP_coeffs, PU_coeffs):
        """Function to add diffusion terms to coefficient arrays"""

        # Calculate the mass imbalance, based on advecting velocities
        imbalance = self._rho*self._grid.Ae*self._Uhe[1:] - self._rho*self.
↪_grid.Aw*self._Uhe[:-1]

        # Calculate the linearization coefficients on pressure
        PP_coeffW = np.concatenate((np.array([0]), -self._rho*self._grid.Aw[1:
↪]*self._dhat[1:-1]/self._grid.dx_WP[1:]))
        PP_coeffE = np.concatenate((-self._rho*self._grid.Ae[:-1]*self._dhat[1:
↪-1]/self._grid.dx_PE[:-1], np.array([0])))
        PP_coeffP = - PP_coeffW - PP_coeffE

        # Calculate the linearization coefficients on velocity
        PU_coeffW = np.concatenate((np.array([-self._rho*self._grid.Aw[0]]), -0.
↪5*self._rho*self._grid.Aw[1:]))
        PU_coeffE = np.concatenate((0.5*self._rho*self._grid.Ae[:-1], np.
↪array([self._rho*self._grid.Ae[-1]])))
```

```
        PU_coeffP = np.concatenate((np.array([0]), PU_coeffW[1:])) + np.
↪concatenate((PU_coeffE[:-1], np.array([0])))

        # Modify the linearization coefficients on the boundaries
        # (velocity only, since pressure is already zero)
        PU_coeffP[0] += PU_coeffW[0]*self._U_west_bc.coeff()
        PU_coeffP[-1] += PU_coeffE[-1]*self._U_east_bc.coeff()

        # Zero the boundary coefficients that are not used
        PU_coeffW[0] = 0.0
        PU_coeffE[-1] = 0.0

        # Add to coefficient arrays
        PP_coeffs.accumulate_aP(PP_coeffP)
        PP_coeffs.accumulate_aW(PP_coeffW)
        PP_coeffs.accumulate_aE(PP_coeffE)
        PP_coeffs.accumulate_rP(imbalance)
        PU_coeffs.accumulate_aP(PU_coeffP)
        PU_coeffs.accumulate_aW(PU_coeffW)
        PU_coeffs.accumulate_aE(PU_coeffE)

        # Return the modified coefficient arrays
        return PP_coeffs, PU_coeffs
```

Additionally, the linear solver in Lesson5/LinearSolver.py has been updated to work with multiple coefficient matrices, as required for coupled equations.

For pressure, we will also need an extrapolated boundary condition, as shown in the outline below:

[9]:
```
class ExtrapolatedBc:
    """Class defining an extrapolated boundary condition"""

    def __init__(self, phi, grid, loc):
        """Constructor
            phi ........ field variable array
            grid ....... grid
            loc ........ boundary location
        """
        self._phi = phi
        self._grid = grid
        self._loc = loc

    def value(self):
        """Return the boundary condition value"""
        if self._loc is BoundaryLocation.WEST:
            return # Fill in expression
        elif self._loc is BoundaryLocation.EAST:
            return # Fill in expression
        else:
            raise ValueError("Unknown boundary location")

    def coeff(self):
        """Return the linearization coefficient"""
        if self._loc is BoundaryLocation.WEST:
            return # Fill in expression
        elif self._loc is BoundaryLocation.EAST:
            return # Fill in expression
        else:
            raise ValueError("Unknown boundary location")

    def apply(self):
        """Applies the boundary condition in the referenced field variable␣
 ↪array"""
        if self._loc is BoundaryLocation.WEST:
            pass # Fill in expression below
            #self._phi[0] =
        elif self._loc is BoundaryLocation.EAST:
            pass # Fill in expression below
            #self._phi[-1] =
        else:
            raise ValueError("Unknown boundary location")
```

Exercise: Complete the extrapolated boundary condition above.

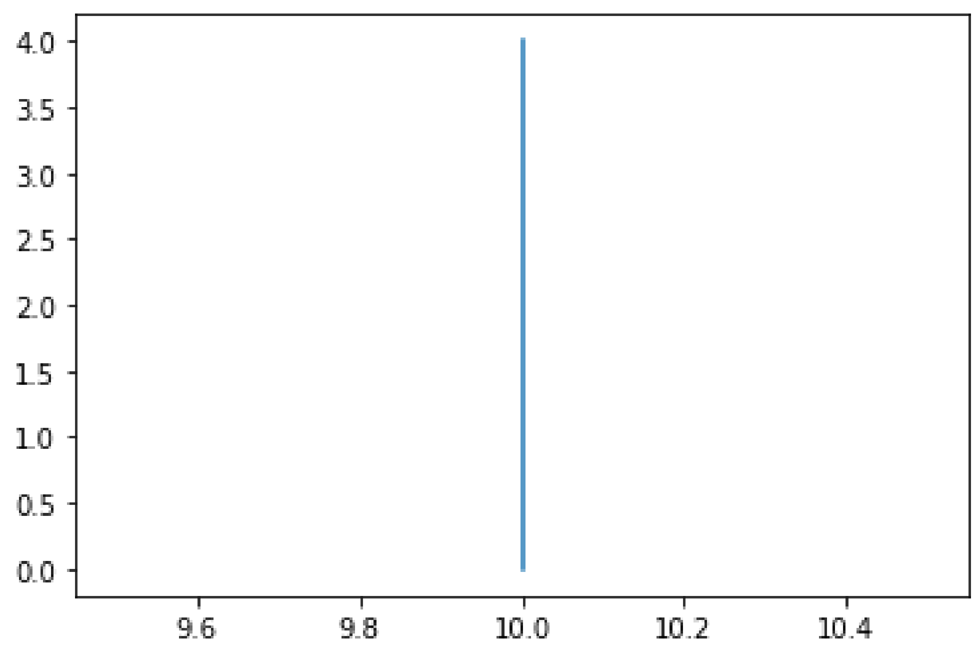
The example below sets up a case with a constant velocity of 10 [m/s] and 0 [Pa]. Once all of the code above is completed, this will accept the initial condition as the correct solution, since there are no drag forces.

[11]:
```
from Lesson5.Grid import Grid
from Lesson5.ScalarCoeffs import ScalarCoeffs
from Lesson5.BoundaryConditions import BoundaryLocation, DirichletBc, NeumannBc
from Lesson5.Models import DiffusionModel
from Lesson5.LinearSolver import solve

import numpy as np
from numpy.linalg import norm

# Define the grid
lx = 4.0
ly = 0.02
lz = 0.02
ncv = 10
grid = Grid(lx, ly, lz, ncv)

# Set the timestep information
nTime = 1
```

```
dt = 1e9
time = 0

# Set the maximum number of iterations and convergence criterion
maxIter = 100
converged = 1e-6

# Define thermophysical properties
rho = 1000
mu = 1e-3

# Define the coefficients
PU_coeffs = ScalarCoeffs(grid.ncv)
PP_coeffs = ScalarCoeffs(grid.ncv)
UP_coeffs = ScalarCoeffs(grid.ncv)
UU_coeffs = ScalarCoeffs(grid.ncv)

# Initial conditions
U0 = 10
P0 = 0

# Initialize field variable arrays
U = U0*np.ones(grid.ncv+2)
P = P0*np.ones(grid.ncv+2)

# Initialize advecting velocity and damping coefficient array
dhat = np.zeros(grid.ncv+1)
Uhe = U0*np.ones(grid.ncv+1)

# Define boundary conditions for velocity
U_west_bc = DirichletBc(U, grid, U0, BoundaryLocation.WEST)
U_east_bc = NeumannBc(U, grid, 0, BoundaryLocation.EAST)

# Define boundary conditions for pressure
#   - Once ExtrapolatedBc is complete, change the boundary condition
#P_west_bc = ExtrapolatedBc(P, grid, BoundaryLocation.WEST)
P_west_bc = NeumannBc(P, grid, 0, BoundaryLocation.WEST)
P_east_bc = DirichletBc(P, grid, 0, BoundaryLocation.EAST)

# Apply boundary conditions
U_west_bc.apply()
U_east_bc.apply()
P_west_bc.apply()
P_east_bc.apply()

# Define the transient model
Uold = np.copy(U)
```

```
transient = FirstOrderTransientModel(grid, U, Uold, rho, 1, dt)

# Define the diffusion model
diffusion = DiffusionModel(grid, U, mu, U_west_bc, U_east_bc)

# Define the advection model
advection = UpwindAdvectionModel(grid, U, Uhe, rho, 1, U_west_bc, U_east_bc)

# Define the pressure force model
pressure = PressureForceModel(grid, P, P_west_bc, P_east_bc)

# Define advecting velocity model
advecting = AdvectingVelocityModel(grid, dhat, Uhe, P, U, UU_coeffs)

# Define conservation of mass equation
mass = MassConservationEquation(grid, U, P, dhat, Uhe, rho,
                                P_west_bc, P_east_bc, U_west_bc, U_east_bc)

# Loop through all timesteps
for tStep in range(nTime):
    # Update the time information
    time += dt

    # Print the timestep information
    print("Timestep = {}; Time = {}".format(tStep, time))

    # Store the "old" velocity field
    Uold[:] = U[:]

    # Iterate until the solution is converged
    for i in range(maxIter):

        # Zero all of the equations
        PP_coeffs.zero()
        PU_coeffs.zero()
        UU_coeffs.zero()
        UP_coeffs.zero()

        # Assemble the momentum equations
        #   Note: do this before mass, because the coeffs are needed to compute␣
 ↪advecting velocity
        UU_coeffs = diffusion.add(UU_coeffs)
        UU_coeffs = advection.add(UU_coeffs)
        UU_coeffs = transient.add(UU_coeffs)
        UP_coeffs = pressure.add(UP_coeffs)

        # Assemble the mass equations
        advecting.update()
        PP_coeffs, PU_coeffs = mass.add(PP_coeffs, PU_coeffs)

        # Compute residuals and check for convergence
        PmaxResid = norm(PU_coeffs.rP + PP_coeffs.rP, np.inf)
        PavgResid = np.mean(np.absolute(PU_coeffs.rP + PP_coeffs.rP))
        UmaxResid = norm(UU_coeffs.rP + UP_coeffs.rP, np.inf)
        UavgResid = np.mean(np.absolute(UU_coeffs.rP + UP_coeffs.rP))
        print("Iteration = {}.".format(i))
        print("  Mass:     Max. Resid. = {}; Avg. Resid. = {}".
 ↪format(PmaxResid, PavgResid))
        print("  Momentum: Max. Resid. = {}; Avg. Resid. = {}".
 ↪format(UmaxResid, UavgResid))
        if PmaxResid < converged and UmaxResid < converged:
            break

        # Solve the sparse matrix system
        dP, dU = solve(PP_coeffs, PU_coeffs, UP_coeffs, UU_coeffs)

        # Update the solutions
        P[1:-1] += dP
        U[1:-1] += dU

        # Update boundary conditions
        U_west_bc.apply()
        U_east_bc.apply()
        P_west_bc.apply()
        P_east_bc.apply()

        # Update the advecting velocities
        advecting.update()
import matplotlib.pyplot as plt
plt.plot(U,grid.xP)
# plt.show()
print(P)
```

```
Timestep = 0; Time = 1000000000.0
Iteration = 0.
  Mass:     Max. Resid. = 0.0; Avg. Resid. = 0.0
  Momentum: Max. Resid. = 0.0; Avg. Resid. = 0.0
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```



Exercise: Add a drag force to the model above to get the pressure to change.

[ ]:
```

```