```python
from ComputeTimeStep import computeTimeStep
from InputVariables import ngc, iniCond
from FixedVariables import gamma, nw, rho, v, rhov, e, bc, xmax, xmin, csound, p, G
import numpy as np

# returns an array with zeros with nx cells plus ngc ghostcells on each end
def make_mesh(xmin, xmax, nx):
    dx = (xmax - xmin) / (nx - 1)
    x = []
    for i in range(ngc):
        x.append(0)
    x.append(xmin)
    for j in range(ngc+1, ngc+nx-1):
        x.append(x[j-1]+dx)
    x.append(xmax)
    for k in range(ngc):
        x.append(0)
    return x

# defines initial conditions for 'shock', the shock tube problem and 'acoustic', the linear acoustic wave problem.
# output =

def initialization(nx):
    x = make_mesh(xmin, xmax, nx)
    w = [[] for m in range(nx+ngc*2)]
    if iniCond == 'shock':
        for i in range(0, ngc + nx//2):
            prim = [8, 0, 8/gamma]
            w[i].append(rho(prim))
            w[i].extend(rhov(prim))
            w[i].append(e(prim))
        for i in range(ngc + nx//2, nx + 2*ngc):
            prim = [1, 0, 1]
            w[i].append(rho(prim))
            w[i].extend(rhov(prim))
            w[i].append(e(prim))
        return w
    elif iniCond == 'acoustic':
        for i in range(ngc, ngc + nx):
            AcouP = 0.1 + 0.001*np.exp(- np.square(x[i]-0.5)/0.01)
            prim = [AcouP*gamma, 0, AcouP]
            w[i].append(rho(prim))
            w[i].extend(rhov(prim))
            w[i].append(e(prim))
```

```python
        return w
    else:
        print('no valid initial condition')
    return

# defines values for ghost cells for an array x with dimension nx + 2*ngc and returns the array x with the values of
# ghostcells added.
def getbc(x):
    if bc == 'fixed':
        return x
    elif bc == 'periodic':
        for i in range(ngc):
            x[i] = x[-ngc-1-i]
            x[-(i+1)] = x[ngc+i]
        return x
    else:
        print('no valid boundary condition')

# gives the eigenvalues as a 3 by nx+2*ngc matrix lam
def get_eigenval(w, nx):
    lam = []
    for i in range(nx + 2*ngc):
        C = csound(w[i])
        V = v(w[i])[0]
        lam.append([V - C, V, V + C])
    return lam

# gives the eigenvector matrix for each point in the mesh based on the the eigenvalue array lam
def get_k(w, nx):
    K = [[[] for j in range(nw)] for k in range(nx+2*ngc)]
    for i in range(nx+2*ngc):
        V = v(w[i])[0]
        C = csound(w[i])
        K[i][0].append(1)
        K[i][0].append(1)
        K[i][0].append(1)
        K[i][1].append(V - C)
        K[i][1].append(V)
        K[i][1].append(V + C)
        K[i][2].append(np.square(V)/2 - C*V + np.square(C)/ G)
        K[i][2].append(np.square(V)/2)
        K[i][2].append(np.square(V)/2 + C*V + np.square(C)/ G)
    return K

# gives the inverse eigenvector matrix for each point in the mesh
```

```python
for the velocity and the speed of sound
def get_kinv(w, nx):
    kinv = [[[] for j in range(nw)] for k in range(nx+2*ngc)]
    for i in range(nx+2*ngc):
        V = v(w[i])[0]
        C = csound(w[i])
        kinv[i][0].append(V / (2 * C) + np.square(V) * G / (4 * np.square(C)))
        kinv[i][0].append(-1 / (2 * C) - V * G / (2 * np.square(C)))
        kinv[i][0].append(G / (2 * np.square(C)))
        kinv[i][1].append(1 - np.square(V) * G / (2 * np.square(C)))
        kinv[i][1].append(V * G / np.square(C))
        kinv[i][1].append(-G / np.square(C))
        kinv[i][2].append(-V / (2 * C) + np.square(V) * G / (4 * np.square(C)))
        kinv[i][2].append(1 / (2 * C) - V * G / (2 * np.square(C)))
        kinv[i][2].append(G / (2 * np.square(C)))
    return kinv

# multiplicates Kinv with the conservative variables (w) in each point, resulting in the diagonal variables
def ConsToDiag(w, nx):
    kinv = get_kinv(w, nx)
    wdiag = [[0 for l in range(nw)] for m in range(nx+ngc*2)]
    for i in range(nx+2*ngc):
        for j in range(3):
            for k in range(3):
                wdiag[i][j] += kinv[i][j][k] * w[i][k]
    return wdiag

# multiplicates K with the diagonal variables (wdiag) in each point, resulting in the conservation variables
def DiagtoCons(w, wdiag, nx):
    K = get_k(w, nx)
    w2 = [[0 for l in range(nw)] for m in range(nx+ngc*2)]
    for i in range(nx+2*ngc):
        for j in range(3):
            for k in range(3):
                w2[i][j] += K[i][j][k] * wdiag[i][k]
    return w2

# computes the fluxes in the first order upwind scheme (based on the sign of the eigenvalues)
def get_flux(wdiag, lam, nx, dx):
    f_upwind = [[0 for l in range(nw)] for m in range(nx+ngc*2)]
```

```python
    for i in range(ngc, nx + ngc):
        for j in range(nw):
            if lam[i][j] > 0:
                f_upwind[i][j] = (lam[i][j] * wdiag[i][j] - lam[i - 1][j] * wdiag[i - 1][j]) / dx
            elif lam[i][j] < 0:
                f_upwind[i][j] = (lam[i+1][j] * wdiag[i + 1][j] - lam[i][j] * wdiag[i][j]) / dx
            else:
                f_upwind[i][j] = 0
    return f_upwind

# computes the conservative variables in the next time step (after dt).
def IntegrateTime(x, nx):
    dx = (xmax - xmin) / (nx - 1)
    w = getbc(x)
    lam = get_eigenval(w, nx)
    wdiag = ConsToDiag(w, nx)
    f_upwind = get_flux(wdiag, lam, nx, dx)
    dt = computeTimeStep(lam, dx, nx)
    wdiag_adv = [[0 for l in range(nw)] for m in range(nx+ngc*2)]
    for i in range(nx + ngc*2):
        for j in range(nw):
            wdiag_adv[i][j] = wdiag[i][j] - dt*f_upwind[i][j]
    w_adv = DiagtoCons(w, wdiag_adv, nx)
    return dt, w_adv

```